INSTITUTO POLITÉCNICO DE BRAGANÇA Escola Superior de Tecnologia e Gestão

UTFPR UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ForkSim: Animation of Programs to Support the Learning of Concurrent Programming

Daniel Augusto Rodrigues Farina

Dissertation presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology - Paraná (UTFPR)

Supervisors:

Prof. Dr. Maria João Varanda Pereira

Prof. Dr. José Carlos Rufino Amaro

Prof. Dr. Rodrigo Campiolo

Bragança

June 2024

INSTITUTO POLITÉCNICO DE BRAGANÇA Escola Superior de Tecnologia e Gestão

UTFPR UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ForkSim: Animation of Programs to Support the Learning of Concurrent Programming

Daniel Augusto Rodrigues Farina

Dissertation presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology - Paraná (UTFPR)

Supervisors:

Prof. Dr. Maria João Varanda Pereira

Prof. Dr. José Carlos Rufino Amaro

Prof. Dr. Rodrigo Campiolo

Bragança

June 2024

Acknowledgment

I would like to express my deepest gratitude to all who have contributed to the completion of this work and to my academic journey.

First, I would like to acknowledge the valuable financial support of Research Center in Digitalization and Intelligent Robotics (CeDRI) and Laboratory for Sustainability and Technology in Mountain Regions (SusTEC). The support of these institutions was fundamental to the development of this project.

I am immensely grateful to the educational institutions IPB (Instituto Politécnico de Bragança) and UTFPR (Universidade Tecnológica Federal do Paraná) for the opportunity to participate in the double degree program. This experience was enriching and opened up new horizons in my academic and professional training.

To my supervisors and co-supervisors, Prof. Maria João Varanda Pereira, Prof. José Carlos Rufino Amaro and Prof. Rodrigo Campiolo, my sincere thanks. Your guidance, patience and knowledge were essential for the completion of this work. I would also like to thank the professors of UTFPR and IPB for the quality of their teaching and for their support throughout this journey.

I could not fail to thank my classmates, specially Miguel Afonso Beckers and Luiz Henrique de Barros de Oliveira, who shared this journey with me, and my friends and family, for their unconditional support and constant encouragement.

I thank God for granting me the strength and wisdom to face challenges and achieve my goals.

Finally, I thank everyone who, in some way, contributed to the completion of this work. Every word of support, every piece of advice and every gesture of encouragement

was fundamental to this achievement. Thank you all very much.

Funding: This work was supported by national funds through FCT/MCTES (PIDDAC): CeDRI, UIDB/05757/2020 (DOI: 10.54499/UIDB/05757/2020) and UIDP/05757/2020 (DOI: 10.54499/UIDP/05757/2020); and SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P /0007/2020).

Abstract

This document discusses the development of *forkSim*, a tool that can be used to support the teaching of system-level programming within the context of Operating Systems classes, by facilitating the comprehension and analysis of the behavior of C codes representing process-based concurrent programs involving *fork* system calls.

The tool builds on two main components. The first is a C code preprocessor, created using language processing techniques. The preprocessor embeds inspectors into the C code before its compilation and execution. In runtime, the inspectors extract relevant data from the actions performed and generate a JSON file. The second component is a web application that generates a visual representation of the program flow based on the JSON file. This visualization incorporates elements from BPMN diagrams and draws inspiration from representations used for many years in OS classes.

The development of *forkSim* faced several technical challenges and involved some design decisions, both documented in this document, along with a discussion of the results achieved. The tool was evaluated in a real classroom environment, and the results suggest that it can be a valuable resource for students and instructors in the context of Operating Systems classes.

Keywords: E-Learning Tool, Operating System, System Programming, Code Instrumentation, Inspector Functions, C, BPMN.

Resumo

Este documento discute o desenvolvimento do *forkSim*, uma ferramenta que pode ser usada para apoiar o ensino de programação de nível de sistema no contexto das aulas de Sistemas Operativos, facilitando a compreensão e análise do comportamento de códigos C que representam programas concorrentes baseados em processos envolvendo chamadas de sistema *fork*.

A ferramenta se baseia em dois componentes principais. O primeiro é um preprocessador de código C, criado utilizando técnicas de processamento de linguagem. Este pré-processador insere inspetores no código C antes de sua compilação e execução. Em tempo de execução, os inspetores extraem dados relevantes das ações realizadas e geram um arquivo JSON. O segundo componente é uma aplicação web que gera uma representação visual do fluxo do programa com base no arquivo JSON. Essa visualização incorpora elementos de diagramas BPMN e se inspira em representações utilizadas há muitos anos em aulas de Sistemas Operativos.

O desenvolvimento do *forkSim* enfrentou vários desafios técnicos e envolveu algumas decisões de design, todas documentadas neste documento, juntamente com uma discussão dos resultados alcançados. A ferramenta foi avaliada em um ambiente de sala de aula real, e os resultados sugerem que ela pode ser um recurso valioso para estudantes e instrutores no contexto de aulas de Sistemas Operativos.

Palavras-chave: Ferramentas E-Learning, Sistemas Operacionais, Programação de Sistemas, Instrumentação de código, Funções de inspeção, C, BPMN.

Contents

List of Figures

Chapter 1

Introduction

Visual representation is a widely used resource to facilitate the understanding of many concepts in the learning process. As images are stored in our long-term memory, learning with visual representations has a significant impact on information retention [1]. The effectiveness of this approach is demonstrated by the quantity of academic work that employs it to explain and understand abstract concepts [2]–[4]. In the teaching of Computer Science and Information Systems, many contents are abstract and complex, making the use of visual representations even more important. Some examples are sorting algorithms, search algorithms, data structures, computer networks, operating systems, distributed systems, among others.

Understanding process-based concurrent programs is challenging for Operating System (OS) students. With the aim of facilitating the teaching of this topic, *forkSim* (fork Simulator) was developed, as an educational tool that generates visual representations of the execution flow of such programs in Linux/Unix systems, the typical laboratory environments used in OS classes. In addition to exploring fundamental concepts about the creation of processes and the unpredictability of concurrent programs, the approach undertaken in the development of this work also addresses other topics, such as language processing and program visualization.

Although there are several tools for the simulation and visualization of program execution, many focus on code debugging, aiming to identify errors, as is the case with GDB

[5] and its Graphical User Interface (GUI) front-ends [6]. These debugging tools offer comprehensive interfaces with various features, but they primarily focus on identifying errors and lack visualizations specifically designed for educational purposes. Another well-known tool is Python Tutor, which shows the variables and memory state during program execution (essential for teaching programming), but it does not support programs with *fork* calls.

Considering the constraints and limitations of all these alternatives, it was develop a new tool, specifically tailored to the needs of OS laboratories. This tool goes beyond the visualization of the execution of a conventional (sequential) program and supports the automatic and dynamic visualization of the execution of process-based concurrent programs.

The *forkSim* project is composed of several parts. Its web platform is used to simulate and visualize the execution flow of C programs Users submit their code through a web application, which the backend stores and manages. The submitted codes undergo preprocessing, where inspector functions are injected into the source code. The backend communicate with an isolated service to execute the preprocessed code. The execution results in a file with logs of the program's behavior.

With the application running, a classroom survey was conducted. The survey involved a questionnaire on the usability of the tool and the understanding of concurrency concepts. An evaluative activity was applied to verify the effectiveness of the tool in the learning process. The results obtained showed a slightly improvement in students' understanding, and the questionnaire showed that students felt more motivated and interested in learning about the subject.

The remainder of this dissertation is organized as follows. Chapter 2 presents the concepts, technologies and tools used, and related works. Following this, chapter 3 describes the development of *forkSim*, covering everything from its conception to implementation. Subsequently, chapter 4 shows a case of the tool and its evaluation. Finally, chapter 5 discusses the conclusions and future works.

Chapter 2

Context and Tools

This chapter introduces concepts for understanding the design of *forkSim*. Section 2.1 covers concurrent and parallel programming, system calls, graph theory, and language processing. Sections 2.2.1 and 2.2.2 discuss the tools tested and used for creating the visualization of the execution graph and performing syntactic analysis of the source code. Section 2.2 details the materials and tools used throughout development. Section 2.3 presents related works, such as *GDB*, *Python Tutor*, and *GraSMa*. The chapter concludes with a summary of the discussed concepts and tools in Section 2.4.

2.1 Background Concepts

The key concepts discussed in this section are essential for understanding the development of the *forkSim* project. These concepts encompass concurrent and parallel programming, system calls like *fork* and *wait/waitpid*, graph theory, language processing, and compilers. They were discussed in the context of the project's design and implementation, highlighting their relevance to the creation of a tool that facilitates the understanding of concurrent program execution.

2.1.1 Concurrent and Parallel Programming

Concurrent and parallel programming are strategies focused on the simultaneous execution of multiple computational tasks [7]. Today, these strategies are integrated into programming languages, libraries, frameworks, and even source code preprocessors, and can be classified as a programming paradigm.

There are two approaches to work with simultaneous tasks: threads and processes. The choice between threads and processes depends on the problem to be solved, as each approach has its own advantages and disadvantages. Some of the main challenges of these strategies include ensuring correct interaction between tasks, coordinating simultaneous access to computational resources, and facilitating communication between tasks.

Although related, concurrent and parallel programming are different. Concurrent programming emphasizes the interaction between tasks, while parallel programming focuses on the actual simultaneous execution of tasks [8]. In both strategies, if the program is executed on a single machine, the OS manages the execution of tasks and can provide real simultaneous execution or not. Parallel programming is often associated with distributed systems, where tasks are executed on different machines connected by a network, such as in the case of web servers, database servers, and high-performance applications.

The study of parallel and concurrent applications is important because it can significantly increase performance. The simultaneous execution of tasks can reduce the execution time of a program, making it more efficient.

2.1.2 System Calls

System calls are a fundamental mechanism in computing that allow programs to request specific services from the OS [9], like access to hardware, creation and execution of new processes, communication with the OS kernel, and processor scheduling, among others.

System calls provide an interface for controlled and secure access to OS resources. In many systems, system calls are made only by user space processes to ensure the security and integrity of the system. However, in some specific systems, privileged system code

also performs system calls, enabling the execution of critical and sensitive tasks.

Among the various system calls available, the *fork* system call holds particular significance in the realm of process management [10]. When a program invokes the *fork* system call, it triggers the creation of a new process, known as the *child* process, which is a copy of the *parent* process. This new process inherits various attributes from its *parent*, including memory, file descriptors, and other resources, but operates independently.

The *fork* system call essentially splits the executing process into two separate execution paths, each running concurrently. The *child* process receives a copy of the *parent*'s address space, including variables, pointers, and instructions, at the moment of the split. However, the two processes then proceed independently, with changes made to memory or resources in one process not affecting the other.

This mechanism is fundamental in various aspects of system operation, including multitasking, parallel processing, and the creation of complex software architectures. By allowing processes to split and execute concurrently, the *fork* system call enables efficient resource utilization and supports the execution of diverse tasks simultaneously.

After a *fork* operation, it is often necessary for the *parent* process to manage the *child* processes it has created. This is where the *wait* and *waitpid* system calls come into play [11]. The *wait* system call makes the *parent* process wait until all of its *child* processes have terminated, at which point it retrieves the termination status of one of the *child* processes. On the other hand, the *waitpid* system call provides more control by allowing the *parent* process to wait for a specific *child* process to terminate, identified by its process ID (PID), or to wait for any child process based on specified criteria. Both *wait* and *waitpid* are used for process synchronization and resource management, ensuring that *child* processes are cleaned up and their resources are released, preventing resource leaks and maintaining system stability.

2.1.3 Graph Theory

Graph Theory is a field of mathematics that studies the relationships between the objects in a given set [12]. A graph is a mathematical structure consisting of a set of elements called vertices (or nodes) and a set of connections between these elements called edges (or vertices). Depending on the application, the edges can be directed or undirected, with or without weights. Graphs are used to solve practical problems and model complex relationships between objects, with applications in various areas such as telecommunications, computing, biology, and logistics, among others.

In this work, graph theory provided a framework for interpreting the actions of concurrent processes through the representation of execution graphs. These graphs, composed of vertices representing program actions and edges denoting transitions between actions, allowed for the visualization and analysis of concurrent program behavior. Leveraging graph theory concepts such as breadth-first and depth-first searches, It enabled the computation of possible sequences of steps within the concurrent programs, facilitating the simulation of execution paths and aiding in the comprehension of concurrency concepts.

2.1.4 Programming Language Processing

Programming language processing is a field of computer science that studies the development of tools and techniques for the analysis and manipulation of programming languages [13]. These tools are used to create of compilers, interpreters, preprocessors, and other softwares that process and transform source code. This section provides an overview of the key concepts and tools used in language processing.

Code Preprocessing

During the compilation of a program, the preprocessor (or precompiler) runs on the input program before it is turned into an executable by the compiler [13]. It is capable of applying transformations such as removal, addition, or even code generation. The number of transformations performed depends on the purpose of the preprocessor. In C

and C++, the preprocessor is responsible for including files from libraries, expanding and substituting macros, and adding or removing code according to compilation conditionals.

The complexity of preprocessors is evaluated based on their understanding of the code. The simplest preprocessors only perform lexical analysis, identifying basic tokens to search for patterns for substitution. More complex preprocessors perform not only lexical analysis but may also conduct semantic analysis, seeking the necessary information to perform specific operations.

Code Instrumentation

Code instrumentation is a process applied to a program before and after its compilation, called static and dynamic instrumentation, respectively. This technique has been applied to C++ programs since 1995, and today it is also used in Java and C# programs [14]. This is feasible because the intermediate format of these programs retains extensive information about the original code. This information, generally used in reverse engineering programs, is referred to as metadata or "information about the information".

Static instrumentation involves modifying the source code or intermediate code before the program is compiled or executed. This means that the additional code for monitoring or analysis is embedded into the program during the build process. In contrast, dynamic instrumentation refers to the insertion of code while the program is running. This approach allows for flexibility and the ability to monitor or modify the program's behavior without changing the original code base. Both static instrumentation techniques are important for tasks such as profiling, debugging, and performance tuning.

Metadata enables the addition, removal, or alteration of program elements while the program is still running. This capability allows for performance analysis, behavioral analysis, and the identification of problems and critical points in a non-intrusive manner. However, the most significant drawback of using code instrumentation is the complexity of its implementation. The lack of specific tools for this purpose requires a deep understanding of low-level details of programming languages to build these functionalities.

Aspect-Oriented Programming

The Aspect-Oriented Programming (AOP) paradigm is an approach to modularize and encapsulate behaviors that are dispersed across different locations in the code [15]. This paradigm provides a mechanism to keep the code more organized and easier to maintain.

AspectJ is a language that implements the AOP paradigm for Java projects. Using this tool, developers can configure actions to be executed at specific points during the application's execution, such as before or after a method call. During compilation, preprocessing modifies the code according to the implemented aspects.

This paradigm is particularly useful for implementing cross-cutting concerns, which are functionalities that affect multiple parts of the code. In the context of this work, the AOP paradigm could be used to insert monitoring instructions into the source code, facilitating the visualization of the program's execution flow. Albeit not used in this project, the AOP paradigm brings insights into the development of tools for code analysis and manipulation.

2.1.5 Compilers

A compiler is a program (or set of programs) that reads another program written in a programming language and translates it into an equivalent program in binary code (if it is a translator it can be transformed in another language) [13]. Throughout this process, called compilation, the compiler performs various analyses and transformations to ensure the correctness and efficiency of the resulting program.

Compilation consists of several steps: lexical analysis, syntactic analysis, semantic analysis, and code generation. Lexical analysis identifies tokens in the source code, such as keywords, identifiers, and numbers. Syntactic analysis verifies the structure of the source code and the relationships between tokens. Semantic analysis ensures the coherence of the source code, identifying type errors, use of undeclared variables, and more. Code generation produces the resulting program in the target language, usually machine code.

There are other programs that assist the compiler, such as the preprocessor, the assembler, and the linker. The preprocessor handles macros, includes files, and removes comments. The assembler translates assembly code into machine code, while the linker combines the object files generated by the compiler into a single executable file.

Compilers can fall into various categories [16], [17]. Traditional compilers for compiled programming languages produce executable programs. However, some compilers generate code for a virtual machine, interpreted by an interpreter, as seen in Java. Additionally, there are compilers that translate code from one programming language to another, such as the TypeScript (TS) to JavaScript (JS) compiler.

This work leveraged the GNU Compiler Collection (GCC) to compile the preprocessed source code into an executable program [18]. During compilation, flags were employed to identify and report potential errors, promoting code quality and correctness.

2.1.6 BPMN

BPMN, is a standardized graphical representation method for modeling business processes [19]. It allows for a clear and systematic illustration of the steps involved in a business process from beginning to end, making the process easier to understand, analyze, and communicate.

In BPMN, flow objects are used to depict the behavior of the process. These include: events, which signify something happening; activities, which are tasks being performed; and gateways, which control the flow based on conditions. These elements are connected by various types of lines, showing the order of activities, the flow of messages between different participants, and associations between elements and additional information. Figure 2.1 shows an example of a BPMN diagram.

The diagram is organized using swimlanes, which help categorize activities based on the responsible parties. Pools represent the different participants in the process, and lanes are subdivisions within pools, indicating specific roles or departments. Artifacts such as data objects, groups, and annotations provide supplementary information, enriching the

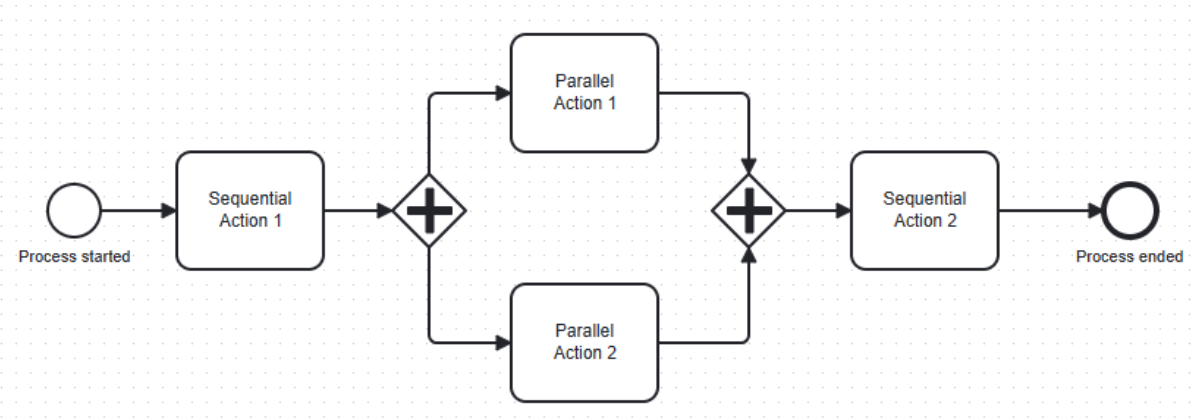


Figure 2.1: Example of a BPMN diagram.

process description without influencing the flow directly.

BPMN serves as a bridge between the design of business processes and their implementation. It is valuable for both business analysts, who design and optimize processes, and technical developers, who implement these processes in information systems. By using a standardized visual language, BPMN eases effective collaboration among stakeholders, ensuring that the designed processes are efficient and aligned with business objectives.

This work draws inspiration from BPMN elements, intersecting the concepts of the "process tree" and "process management flows". This intersection allows for a structured and visual approach to understanding program actions as business processes. Each process in the tree can be interpreted as a BPMN task, with the lifecycle of processes (starting, executing, and terminating) corresponding to BPMN events. *Fork* calls, which create new *child* processes, are akin to BPMN gateways that introduce branching based on conditions.

2.2 Selected Tools

The *forkSim* project was developed using a set of tools selected based on the requirements of the project. Table 2.1 presents the main frontend and backend tools utilized in the project. *TS* is employed on the frontend to ensure static typing [20], complemented by the *React JS* framework for constructing the user interface [21]. Additionally, the *Chakra UI* library aids in developing interface components [22], while *Konva* facilitates

the manipulation of graphic elements, particularly in interactive simulations [23]. *Axios* serves the purpose of making HyperText Transfer Protocol (HTTP) requests [24].

On the backend side, *Python* serves as the primary language [25], complemented by the *FastAPI* framework for rapid and efficient Application Programming Interface (API) development [26]. Additional tools include *Databases* for database access [27], *Pydantic* for data validation [28], and *Uvicorn* as the ASGI server [29].

Table 2.1: Main tools of the frontend and backend of the *forkSim* project.

Frontend	**Backend**
TypeScript v5.0.2	Python v3.12
React JS v18.2.0	FastAPI v0.104.1
Chakra UI v2.7.0	Databases v0.8.0
Konva v9.2.0	Pydantic v2.4.2
Axios v0.24.0	Uvicorn v0.14.0

Table 2.2 provides a list of additional tools utilized in the project. *PostgreSQL* was selected as the Database Management System (DBMS) for storing and managing relational data [30]. For code preprocessing, the project employed *Python Lex-Yacc (PLY)* first [31], and later, *Tree-sitter* for more advanced syntactic analysis [32]. *Docker* and *Docker Compose* were used for containerization, offering isolated and consistent environments for development and deployment [33], [34]. *Figma*, a collaborative design tool, was utilized for creating sketches [35], while Graphviz was employed for visualizing graphs [36].

Table 2.2: Main tools used in the *forkSim* project.

Database	**Code Preprocessors**	**Containers**	**Others**
PostgreSQL v16.2	Tree-sitter v0.20.2	Docker v20.10.17	Figma
	PLY v3.11	Docker Compose v2.6.1	Graphviz

2.2.1 Visualization Tools

In this work, various tools were tested in the creation of graphs to facilitate the visualization of program execution flows. Each tool offers unique features and capabilities suited

to different aspects of graph creation and manipulation.

Graphviz stands out as an open-source solution that employs a Domain-Specific Language (DSL) to generate easily usable graphical visualizations [36]. Widely recognized for its ability to produce static graphs for diverse diagram types, including class diagrams and networks, *Graphviz* provides a straightforward approach to graph generation.

Konva is an open-source JS framework tailored for working with the HTML5 canvas [23]. By leveraging the canvas element, *Konva* enables the development of web applications that display graphics and animations independently of HTML standards. Its feature-rich framework supports high-performance animations, transitions, nested nodes, and event handling across desktop and mobile environments.

D3.js, an open-source JS library for data visualization, offers a low-level approach built on web standards [37]. Renowned for its flexibility in creating data-driven dynamic charts, *D3.js* has received numerous awards for its achievements. Its versatility makes it a powerful tool for crafting interactive and visually appealing graphs.

Additionally, the Scalable Vector Graphics (SVG) language, maintained by the World Wide Web Consortium (W3C), provides a markup language for describing two-dimensional graphics and images [38]. Alongside a set of graphics-related instructions, SVG offers a comprehensive platform for creating scalable and interactive vector graphics on the web. The combination of these tools empowers developers to produce visually compelling and informative graphs for various applications.

Table 2.3 summarizes the comparison of the tools in terms of performance, flexibility, and complexity. While *Graphviz* excels in performance and simplicity, it lacks the flexibility and interactivity of the other ones. *D3.js* offers high performance and flexibility but requires a higher level of knowledge to be used effectively. *Konva* provides a balance between performance and flexibility, making it suitable for a wide range of applications. Finally, *SVG* offers high performance and flexibility with low complexity, making it an excellent choice for creating graphics. In this work, *Konva* was chosen first for its balance between performance and flexibility and then updated by *SVG* for its simplicity and compatibility with the project's requirements.

Table 2.3: Comparison of Visualization Tools

Library	Performance	Flexibility	Complexity
Graphviz	High	Low	Very Low
D3.js	Very High	High	High
Konva	Medium	High	Medium
SVG	High	High	Low

2.2.2 Syntax Analysis Tools

In this work, several tools were tested to build the preprocessor to facilitate parsing and manipulating the C code. These tools were chosen based on their capabilities and how well they fit the project's requirements. This section provides an overview of each tool, explaining its purpose and capabilities.

PLY [31] is an implementation of the compiler-building tools *lex* and *yacc* for Python. It adheres closely to the original tools' functionality, supporting parsing based on Look-Ahead Left-to-Right (1 token) (LALR(1)) , input validation, error reporting, and diagnostics. Developed for use in the Introductory Compiler Course at the University of Chicago in 2001, PLY provides a robust framework for lexical analysis and parsing in Python.

Lark [39] is a parsing toolkit for Python.It is capable of parsing any context-free grammar and offers support for two parsing options: LALR(1) and Earley. The LALR(1) option is fast and lightweight, comparable to PLY, while the Earley option is more flexible and can handle more complex grammars, albeit at a slower speed. This makes Lark a versatile choice for a variety of parsing tasks.

Another Tool for Language Recognition (ANTLR) [40] is a Left-to-Right, Leftmost derivation with arbitrary lookahead (LL(*)) parser generator used for reading, processing, executing, or translating structured or binary text. Developed since 1989, it supports various programming languages, including Java, C#, Python, and JS. ANTLR is widely used and comes with numerous ready-to-use grammars, tutorials, and usage examples, making it a comprehensive tool for language processing.

Tree-sitter [32] is a parser generator tool and an incremental parsing library. It can

build a concrete syntax tree for a code file and update this tree as the file is edited. It is versatile enough to parse any programming language, fast enough to parse at each keystroke in a text editor, robust enough to provide useful results even in the presence of syntax errors, and dependency-free, allowing it to be embedded in any application. Tree-sitter's ability to provide real-time parsing feedback makes it particularly useful for interactive development environments.

2.3 Related Works

This section presents related works that have influenced the development of the *forkSim* project. Three tools are discussed: GDB, Python Tutor, and GraSMa.

2.3.1 GDB

GDB is a powerful tool for debugging and analyzing programs at a low level, providing detailed insights into program behavior, useful for experienced developers. It is used to analyze and fix issues in programs written in languages such as C and C++ [5], [6]. It allows for detailed inspection of what a program is doing at various stages of its execution.

When a developer runs a program using GDB, the program can be paused at specific points to examine its state. This involves checking the values of variables, understanding the flow of control through the code, and seeing how the program's state changes over time. If a program crashes, GDB can show the series of function calls that led to the crash, helping the developer to pinpoint the exact cause.

GDB also enables step-by-step execution, which means a developer can run the program one line at a time to see how each instruction affects the program. This is useful for identifying logical errors and understanding complex code behavior. Figure 2.2 shows an example of the GDB interface, illustrating how the program's state can be inspected during execution.

Furthermore, GDB allows modification of program variables during execution, which is helpful for testing how different inputs affect the program's behavior without needing

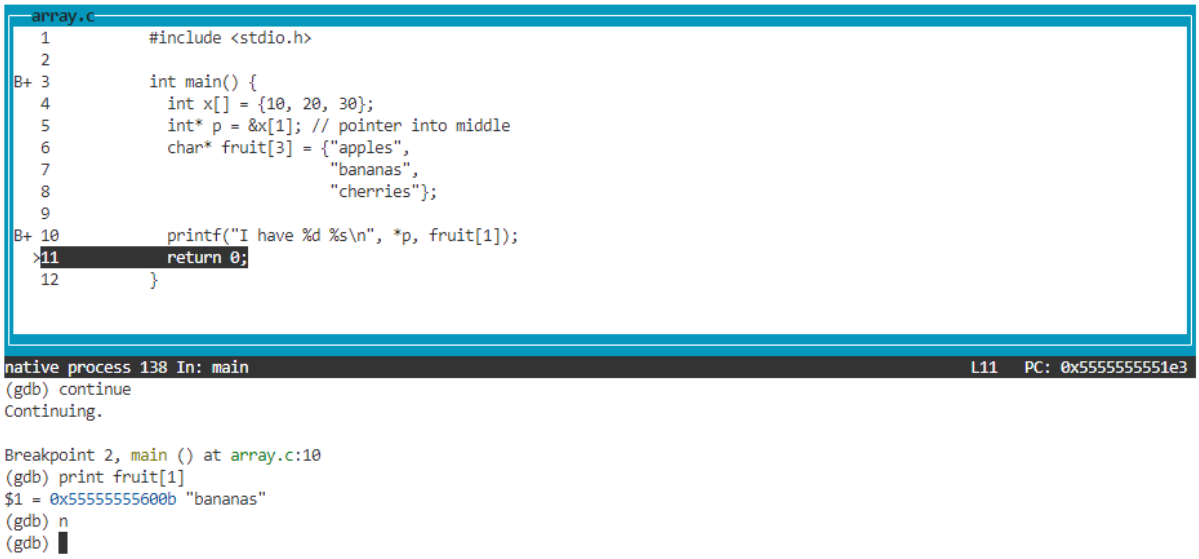


Figure 2.2: Example of GDB interface.

to recompile it each time.

In essence, GDB provides a detailed, interactive way for developers to understand and correct the behavior of their programs, making it an important tool in the software development process.

2.3.2 Python Tutor

Python Tutor is a tool ideal for visualizing code execution and understanding programming concepts in a beginner-friendly manner. It is a web-based tool designed to help users learn programming by visualizing code execution [41]–[43]. It allows users to write and run Python code in an interactive environment where they can see step-by-step how the code is executed. As the code runs, Python Tutor visually represents the changes in variables, data structures, and control flow, making it easier for users to understand how their code works.

This tool is particularly useful for beginners and educators, as it provides a clear and intuitive way to see what happens inside the computer as a program runs. Users can observe how values change over time and how different parts of the code interact.

This helps in debugging and learning programming concepts more effectively. Figure 2.3 shows an example of the Python Tutor interface, illustrating how the code execution is visualized.

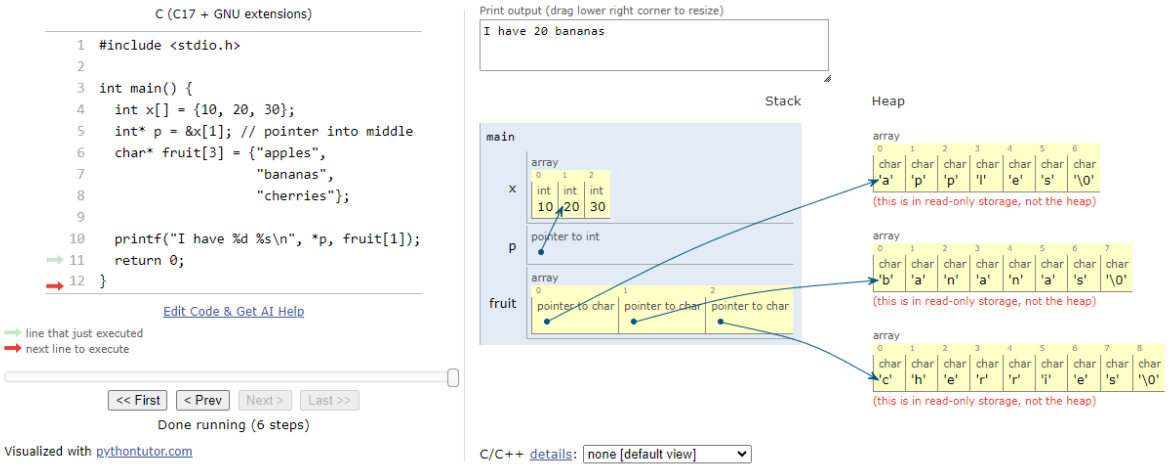


Figure 2.3: Example of Python Tutor interface.

Python Tutor supports multiple programming languages, including Python, Java, JS, C, C++, and Ruby, offering a broad range of learning opportunities. By visualizing the code execution, it bridges the gap between writing code and understanding the underlying processes, enhancing the learning experience.

2.3.3 GraSMa

GraSMa, which stands for Graphical Simulator of Mathematical Algorithms, is a tool designed to assist mathematics teachers and students in understanding numerical algorithms through visualization [44]. Developed using Java technologies, GraSMa works by instrumenting the original code of mathematical algorithms with additional functions called inspector functions. This instrumentation process automatically adds these functions to the code, enabling the capture of detailed information about the algorithm's execution.

During the execution of an algorithm, the inspector functions collect data on how the algorithm operates. This data is recorded in XML files, which serve as a structured format for storing the execution details. A separate Java application then processes these XML

files, transforming the recorded data into graphical representations. These visualizations help users to see and comprehend the behavior and properties of the mathematical objects involved in the algorithms.

GraSMa is a tool designed to help students and teachers understand mathematical algorithms through visualization, providing insights into the behavior of numerical algorithms. It covers the same approach as the *forkSim* project to collect data during the execution of programs.

2.3.4 Discussion

These tools cannot solve the same problem as the *forkSim* project, as they lack the ability to visualize the execution of *fork* system calls. GDB is a powerful tool for debugging, but it does not trace each step for all processes created by *fork* system calls. Python Tutor is a tool for beginners and educators, and it does not support the use of *fork* system calls. GraSMa is a tool for understanding mathematical algorithms, so it has a different focus from the *forkSim* project, albeit it uses a similar approach to collect data during the execution of programs.

2.4 Final Considerations

This chapter introduced the theoretical concepts and tools that underpin the development of the *forkSim* project. Some of them are directly related and others are discussed solely for the purpose of providing a broader understanding of the project's context. For example, while AOP was not used in the project, it is alternative approaches to solving the same problem to insert the inspector functions. In addition, related works are discussed and used to justify the creation of *forkSim*. In the next chapter, the specific details concerning the development of the *forkSim* platform are presented and discussed.

Chapter 3

Development

This chapter describes the stages of development for the *forkSim* project. Section 3.2 describes the developed application, including the C code preprocessors, interface sketch and prototype, frontend, backend, application architecture, and containerization. Finally, Section 3.3 provides an overview of the methodology used to develop the project.

3.1 Introduction

The name *forkSim* was created by joining the words "fork" and "simulation". The word "fork" refers to the fork system calls in Unix systems, used to create processes. When a process invokes a fork system call, the OS creates a copy of it, called the *child* process, while the original is referred to as the *parent* process. From that point on, the *parent* and *child* processes follow independent paths. The word "simulation" describes the project's intention to simulate the execution of concurrent process-based programs, creating an interactive and educational environment for teaching operating systems.

The development of *forkSim* was carried out in stages, with each stage consisting of a set of activities that had to be completed to achieve specific goals. The main objective is to create an application that visualizes graphs of the execution of programs that include fork system calls, aiding in the teaching of operating systems. To achieve this objective, three questions needed to be answered:

1. How can the required information to build graphs of the execution of C programs be collected?

2. What should the visual representation of these graphs look like?

3. How can these graphs be effectively visualized?

To answer the 1st question, concepts related to compilers, language processing, lexical and syntactic analysis phases, and code inspection tools were studied. Instrumentation and code inspection were identified as potential solutions. The *ANTLR*, *PLY*, *Lark*, and *Tree-sitter* tools were tested for transforming C programs by inserting inspection functions to collect information about program execution. The 2nd question was addressed during the application's development, focusing on usability and user experience. This was verified by comparing the initial interface sketch to the final application. For the third question, graph visualization tools such as *Graphviz*, *D3.js*, *Konva*, and *SVG* were tested.

This project involved developing two C code preprocessors. Preprocessor 1 was a proof-of-concept for automatically inserting code inspectors using Python's *PLY* library. Preprocessor 2 was an updated version that more comprehensively recognized the syntactic structure of the C language using the *Tree-sitter* library.

To represent the graphs of C program execution, an application interface sketch was created. This sketch was used to develop a *React* prototype to validate and test the graph construction tools. Based on the prototype, the *Konva* library proved to be the most flexible option for creating graphs. Finally, an *SVG* update was implemented to improve the application's performance.

To integrate these functionalities, a backend for managing C program execution was developed. This allows users to submit programs to be processed and viewed in the application, eliminating the need for manual preprocessing and execution. *Docker Compose* was used to create a Docker container environment for the application, ensuring consistent execution across different machines.

3.2 Application Development

The main components of the *forkSim* project are the C code preprocessor and the simulator of C program execution. The preprocessor is responsible for transforming the original C source code into a new code enriched with inspection functions. The simulator generates the execution graph of this code using the information collected by the inspectors during the program's execution. These parts are connected in a client-server architecture, where the frontend communicates with the backend through a REST API.

The backend was created to manage the execution of C programs. It receives the original C source code, applies the preprocessor to transform it, and manages the execution of the program in an isolated environment. The results are stored in a PostgreSQL database and made available to the frontend through a REST API.

The frontend is capable of displaying the execution graph of the C program. It provides a user interface for submitting C programs, listing and controlling the parameters for program execution, and visualizing the execution graph.

The next sections dive into the development process of the application. Section 3.2.1 explains the C code preprocessors created. Section 3.2.2 covers the interface sketch and prototype. The frontend is explained in Section 3.2.3. Section 3.2.4 details the backend.

3.2.1 C Code Preprocessor

To be possible to later visualize the effective execution flow of an instance of a process-based concurrent C program, the original source code must be modified such that the relevant actions performed by the program are registered in runtime. In essence, this is an application of *code instrumentation*, a technique commonly used for program visualization [44]–[46]. The main idea is to annotate the code with inspection functions (inspectors), making it possible to extract static and dynamic information from the program execution.

To extract the syntax tree from the C language, a Python program was developed using the PLY library [31] as a proof of concept. Others tools were also tested, such as Lark [39], ANTLR [40], and Tree-sitter [32]. Later, the Tree-sitter library was chosen

for its ability to recognize the syntactic structure of the C language, providing a more comprehensive and robust solution compared to the PLY library.

A set of inspection functions was created to obtain data from the actions considered relevant to be visualized. Both preprocessors include the same set of inspection functions, which are inserted into the C source code. These functions are next listed and described:

- `inspector_function_enter()`: added at the beginning of each function;
- `inspector_function_exit()`: added to the end of each function;
- `inspector_function_call()`: added before invoking a function;
- `inspector_variable_declare()`: added after declaring any variable;
- `inspector_variable_assign()`: added after updating any variable;
- `inspector_condition()`: added over any conditional in `if`, `while`, `do while`, and `for` statements;
- `inspector_exit()`: replaces the `exit()` function call;
- `inspector_printf()`: replaces the `printf()` function call;
- `inspector_fork()`: replaces the `fork()` function call;
- `inspector_wait()`: replaces the `wait()` function call;
- `inspector_waitpid()`: replaces the `waitpid()` function call.

The preprocessors are replaceable components. They are used in the processes of transforming the original C source code into a new version enriched with inspection functions. Figure 3.1 illustrates the sequence of steps through which the original C source code flows, starting from its instrumentation, and ending in the visualization of its execution.

The original C code is processed using the Preprocessor 1 or 2 to generate an in-memory representation of its structure. This representation is transformed into a new version of the code enriched with inspection functions. The instrumented code is executed, and the inspectors register the actions performed in JSON files. To avoid race conditions if all processes shared the same JSON file, a distinct temporary output file is created for

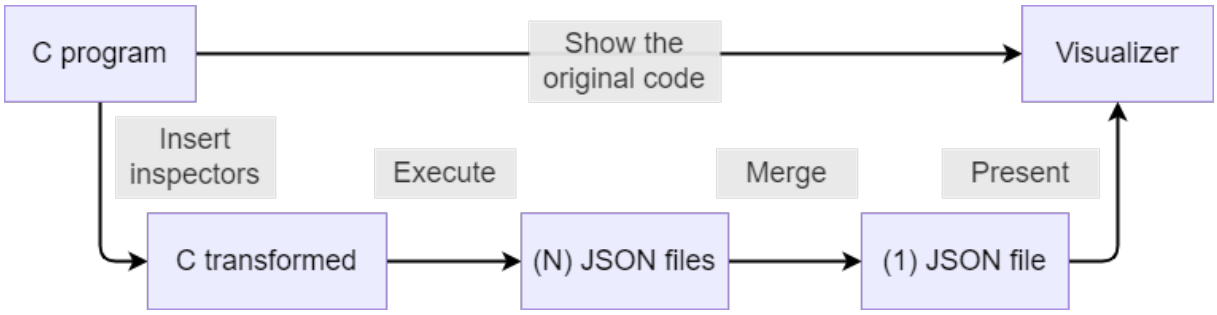


Figure 3.1: Flow of a concurrent program in *forkSim*: instrumentation, execution and visualization.

each OS process during the program's execution. Lastly, the various (N) JSON files are merged into a single file for generating the program visualization.

The JSON file generated has the structure shown in Figure 3.2. Each record includes:

- *time*: time in microseconds at which the action was performed;
- *depth*: depth in the process tree of the program of the issuing process;
- *pid*: process id of the issuing process.
- *parent_pid*: the process id of the parent process of the issuing process;
- *line*: source code line number corresponding to the action;
- *type*: type of the action being performed.
- *function*: the name of the function where the action occurred;
- *payload*: a property to store dynamic values related to the action (For example, *payload* stores the id of the child process in a *fork* call).

On a multi-core machine is possible for several actions in different processes generated by the same program to occur at the same time. This situation is very unlikely in practical terms when using the millisecond time unit. This implies that actions can be aligned sequentially in time even if they are simultaneously, as in single-core systems. The characteristic implicit in Figure 3.2 is presumed to simplify the generation of combinations of possible actions shown in the web interface. Here, the focus lies on the causal relationship between actions, as in the study of logical clocks in Distributed Systems [47].

```
type ProcessActionFile = {
  time: number
  depth: number
  pid: number
  parent_pid: number
  function: string
  type: 'fork_enter' | 'fork_exit'
      | 'function_enter' | 'function_exit' | 'function_call'
      | 'condition' | 'printf' | 'wait' | 'waitpid' | 'exit'
      | 'variable_assign' | 'variable_declare'
  line: number
  payload: any
}
```

Figure 3.2: ProcessActionFile type of each array entry in the JSON file.

Both preprocessors follow the same steps for transforming the original C source code:

1. Reading the content of the input file;
2. Parsing the content using one of the C code preprocessors;
3. Adapting the output of the parser to the project tree representation;
4. Applying transformations to the project tree;
5. Converting the project tree back to a string;
6. Writing the transformed content to the output file.

This sequence of steps is straightforward, with the exception of the third step. In Preprocessor 1, the *PLY* library does not provide a built-in tree representation. Instead, it offers a bottom-up approach to traverse the parsed content, requiring the developer to manually create a tree representation if needed.

In Preprocessor 2, the *Tree-sitter* library provides a tree representation of the parsed content. To ensure consistency in code logic between the two preprocessors when applying

transformations, a middle layer was created. Furthermore, this intermediate representation facilitates the debugging process by allowing the output of a JSON version of the tree and the transformations applied to it.

Preprocessor 1

The first version of the C code preprocessor was developed as a command-line tool in Python using the *PLY* library. Its grammar was specifically tailored to meet the project's requirements and served as a proof of concept for automatically inserting code inspectors into C source code. The grammar designed for this preprocessor is presented in Listing 3.1. It is a simplified version of the C language grammar, encompassing the essential features necessary to support the code examples and typical use-cases of the OS classes. This grammar successfully recognized all the examples presented in the classroom with some minor modifications to the code. Notable changes include separating variable declarations with assigned values onto different lines and omitting calls to the *scanf* function, though this was not due to a limitation of the grammar, but rather for practical reasons related to the code execution. Other characteristics and limitations of this grammar include:

- It strictly recognizes functions with a return type of *int* and arguments of type *int*;
- Pre-increment, pre-decrement, and post-decrement operations are not included;
- Only `int` is recognized as a keyword for data types; all other types are identified as identifiers;
- Structures defined using *struct*, *union*, and *enum* are not identified;
- Variable assignment is limited to the assignment operator =; operations with other assignment formats (+=, -=, etc.) are not recognized.

```
*program : program_declarations
program_declarations : program_declarations program_declaration
                     | program_declaration
program_declaration : include
                    | function
include : INCLUDE INCLUDE_PATH
```

```
        | INCLUDE STRING

function_enter : '{'
*function : INT ID '(' arguments ')' function_enter statements '}'
arguments : arguments ',' argument
          | argument
          |
argument : INT ID

statements : statements statement
           | statement
statement : expression ';'
          | declaration ';'
          | if_statement
          | while_statement
          | do_while_statement ';'
          | for_statement
          | return_statement

expression : function_call
           | literal
           | reference
           | assign
           | unary_expression
           | binary_expression
           | post_increment
           | parenthesis
unary_expression : '-' expression
                 | '!' expression
parenthesis : '(' expression ')'
*post_increment : ID PLUS_PLUS

binary_expression  : expression '+' expression
                   | expression '-' expression
                   | expression '*' expression
                   | expression '/' expression
                   | expression '%' expression
                   | expression '<' expression
                   | expression '>' expression
                   | expression EQ expression
                   | expression NE expression
                   | expression LTE expression
                   | expression GTE expression
                   | expression AND expression
                   | expression OR expression
```

```
*assign : ID assign_symbol expression
assign_symbol : '='
              | PLUS_ASSIGN
reference : '&' ID

literal : string
        | integer
        | id
string : STRING
integer : INTEGER
id : ID

*function_call : ID '(' ')'
               | ID '(' expression_list ')'
expression_list : expression_list ',' expression
                | expression
*if_statement : IF '(' expression ')' '{' statements '}'
              | IF '(' expression ')' '{' statements '}'
                ELSE '{' statements '}'
*do_while_statement : DO '{' statements '}' WHILE '(' expression ')'
*while_statement : WHILE '(' expression ')' '{' statements '}'
*for_statement : FOR '(' expression ';' expression ';' expression ')'
                 '{' statements '}'
*return_statement : RETURN expression ';'

*declaration : type declaration_list
type : INT
     | ID
declaration_list : declaration_list ',' var
                 | var

var : pointer ID array
pointer : pointer '*'
        |
array : array '[' INTEGER ']'
      | array '[' ']'
      |
```

Listing 3.1: C language grammar.

The Listing 3.1 includes rules expressed in Backus-Naur Form (BNF) notation [13] for various aspects of the language, such as: the initial rules of the parsing process; function declarations and their arguments; the structure of declarations in general; the structure of expressions in general; binary expressions; variable assignments, references, and literals;

function calls and control statements; and variable declarations. The rules marked with "*" are rules that were used to insert inspection functions into the original code:

- `inspector_function_enter()`: to process the `function` grammar rule;
- `inspector_function_exit()`: to process the `return_statement` and `function` grammar rules when there is no return;
- `inspector_function_call()`: to process the `function_call` and `assign` grammar rule;
- `inspector_variable_declare()`: to process the `declaration` grammar rule;
- `inspector_variable_assign()`: to process the `assign` and `post_increment` grammar rule;
- `inspector_condition()`: to process the `if_statement`, `while _statement`, `do_while _statement` and `for_statement` grammar rules;
- `inspector_exit()`: to process the `function_call` grammar rule;
- `inspector_printf()`: to process the `function_call` grammar rule;
- `inspector_fork()`: to process the `function_call` grammar rule;
- `inspector_wait()`: to process the `function_call` grammar rule;
- `inspector_waitpid()`: to process the `function_call` grammar rule.

Other transformations were applied to the code to integrate the inspection functions. One of them consists of including the inspector library at the beginning of the code during the processing of the `program` grammar rule. Another modification is the adaptation of the `for` statement, replacing it with the use of `while` when processing the `for_statement` grammar rule. This adaptation allows the processing of internal `for` statements, such as counter initialization, loop condition and counter update. Each of these inspection functions is responsible for sending data from these actions to an output file per each running process and for updating the state of the inspector as necessary.

Preprocessor 2

Similar to the first preprocessor, the second C code preprocessor was developed as a command-line tool in Python. However, it utilized the *Tree-sitter* library to recognize the syntactic structure of the C language. The primary advancement of this preprocessor was its ability to offer a more comprehensive and robust recognition of the C language compared to the version using the *PLY* library. Since there was no need to manually define a grammar, the implementation of transformations was simpler and more direct.

The *Tree-sitter* library provides functionality for navigating and manipulating the syntax tree generated from the source code. However, the library's functions are low-level and not intuitive for transformations. To streamline the implementation of transformations, an external abstraction layer was developed, represented by the *SyntaxNode* class, as demonstrated in Listing 3.2. This class also enables the conversion of the syntax tree to a JSON format, simplifying the visual inspection of the syntax tree structure.

```
class SyntaxNode:
    type: str
    line: int
    column: int
    text: Optional[str]
    children: list['SyntaxNode']
```

Listing 3.2: Project Tree representation.

The development and testing of the second preprocessor involved a series of steps to guarantee its functionality and enhance its capabilities. Initially, tests were designed to verify that the new preprocessor operated as consistently as the first preprocessor. The examples used in Preprocessor 1 were reused to validate the initial stages of development. To further evaluate the new preprocessor's ability to handle more complex syntax, a diverse set of C code samples from a public GitHub repository was used [48]. Additionally, efforts were made to not only replicate the behavior of the first preprocessor but also to address its limitations. By doing so, the overall scope of syntax analysis was broadened, leading to a more robust and capable preprocessing tool.

3.2.2 Sketch and Prototype of the Interface

Considering the application project from the user's perspective, a web interface sketch was created using the Figma tool. This sketch, illustrated in Figure 3.3, defined various layout components, including a bottom panel, navigation controls for switching between actions, and a rudimentary graph display. This graph representation already was still far from the kind of representations that has been used in OS classes at IPB, for many years (see Figure 3.4), and which was aimed to achieve.

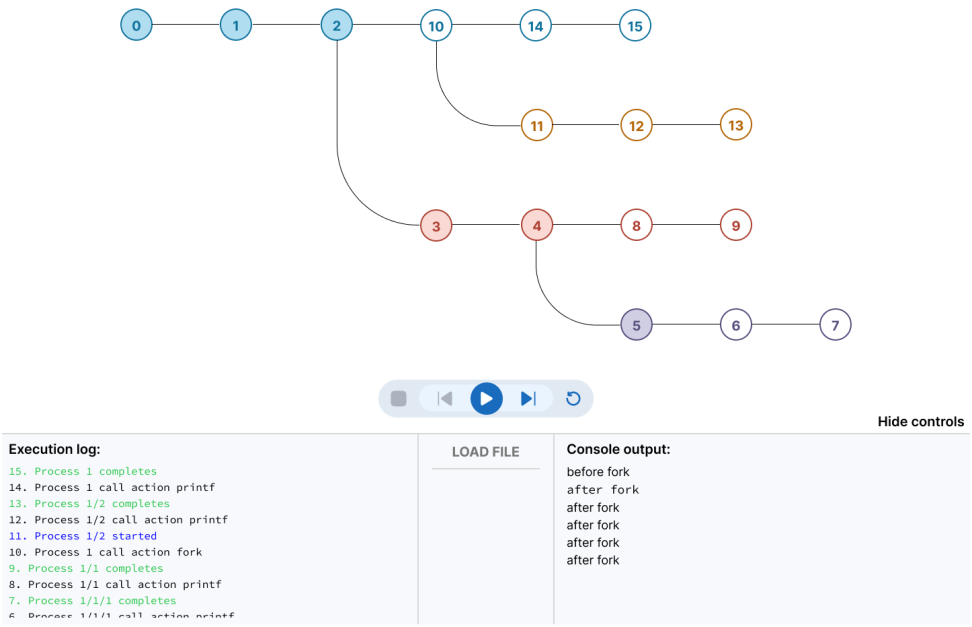


Figure 3.3: Interface sketch of the application.

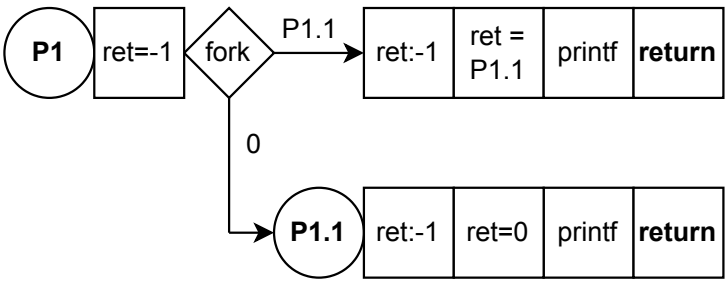


Figure 3.4: Graph representation used in OS classes for the "program1.3.c" program.

These design choices sparked discussions regarding the usability and purpose of the application. The intentionally minimalist nature of the sketch aimed to address initial questions about the application's aspects and provide a minimal representation of what was expected.

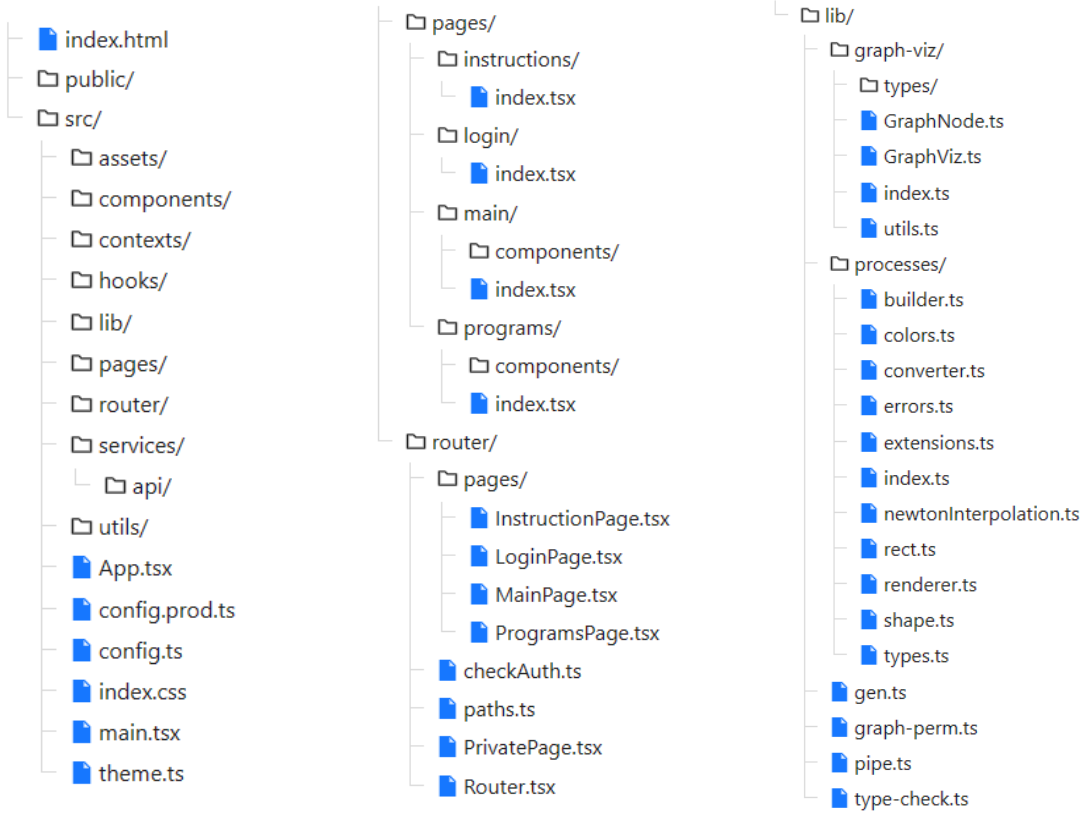
Based on this sketch, a prototype was developed to validate and test the tools for building graphs. The outcome of this prototype raised several questions about the users' understanding of how to interpret the presented graph. Addressing these questions became crucial in selecting the appropriate graph visualization tool.

While the Graphviz library, which presents graphs using the DOT notation, was capable of creating and displaying graphs that closely matched expectations, it lacked interactivity and mechanisms for fine customization. Although the D3.js library offered greater flexibility, its complexity and the requirement for extensive knowledge made it impractical for the project. Conversely, the Konva library, while not primarily focused on graph visualization, provided the ideal combination of flexibility and ease of use, making it the optimal choice for the application.

3.2.3 Frontend

The frontend of a software is responsible for presenting the visual interface of the application and allowing user interaction with the system. In the *forkSim* project, the primary objective of the frontend is to allow users to view and interact with the execution graphs of C programs. This section presents the development of the frontend, including the architecture of the application, the structure of components, integration with the backend, and the implementation of graph visualization.

The frontend development was conducted using the TS language, an extension of JS that incorporates static typing into the code. Key libraries utilized include React JS, Chakra UI, Konva, and Axios. React JS facilitated component declaration and state management, while Chakra UI provided style guidelines and a wide range of customizable components, streamlining frontend development. Axios facilitated communication



(a) Root (b) Pages (c) Lib

Figure 3.5: Directory trees of the frontend.

between the frontend and backend by handling HTTP requests, and Konva was employed for graph creation and visualization.

Following a component-based architecture, each component of the frontend is responsible for a specific aspect of the interface. Figure 3.5 illustrates the directory trees of the frontend from three perspectives: the root directory tree (Figure 3.5a); the pages directory tree (Figure 3.5b); and the libraries directory tree (Figure 3.5c).

Figures 3.5a and 3.5b provides an overview of the project's structure. Within the *components* folder reside reusable components such as buttons, inputs, and modals. Meanwhile, the *pages* folder holds application pages, including the instructions page, login page, program listing, and the main page for graph visualization. External services are

stored in the *services* folder, with backend communication kept in the *api* subfolder. Additionally, application routes and access control are defined in the *routes* folder, while the *routes/pages* directory handles navigation logic and backend communication.

The structure of the project's libraries is illustrated in Figure 3.5c, and includes:

- the *graph-perm* library: provides the *GraphPerm* class, which facilitates depth-first searches with the ability to prune branches; it was designed to analyze the execution flow of a C program and identify feasible execution sequences by eliminating infeasible paths.
- the *gen* library: provides the *Gen* class, which offers extensions for generator manipulation, specifically tailored for handling the results of the *GraphPerm* search;
- the *pipe* library: provides the *Pipe* class, which enables function composition, promoting code readability and facilitating the implementation of algorithms in the application;
- the *type-check* library: provides functions for JSON object integrity verification. This ensures data integrity on the frontend:
- the *graph-viz* library: provides the *GraphViz* class, which enables the generation of graphs in DOT notation (Graphviz), used for graph inspection;
- the *processes* library: provides the *Processes* class, which offers access to various classes for graph generation and rendering; this is the main library of the frontend; it includes the implementation of data conversion from the backend to the frontend format (*converter.ts*), definition of graph shapes (*shape.ts*), construction of graph shapes (*builder.ts*), and rendering utilizing the Konva library (*renderer.ts*); additionally, it incorporates information on colors (*colors.ts*) and data types (*types.ts*), along with auxiliary files housing functions or classes pertinent to the library's operations.

The main page of the application for visualizing execution graphs is shown in Figure 3.6. It includes several components identified by the numbers of the items below:

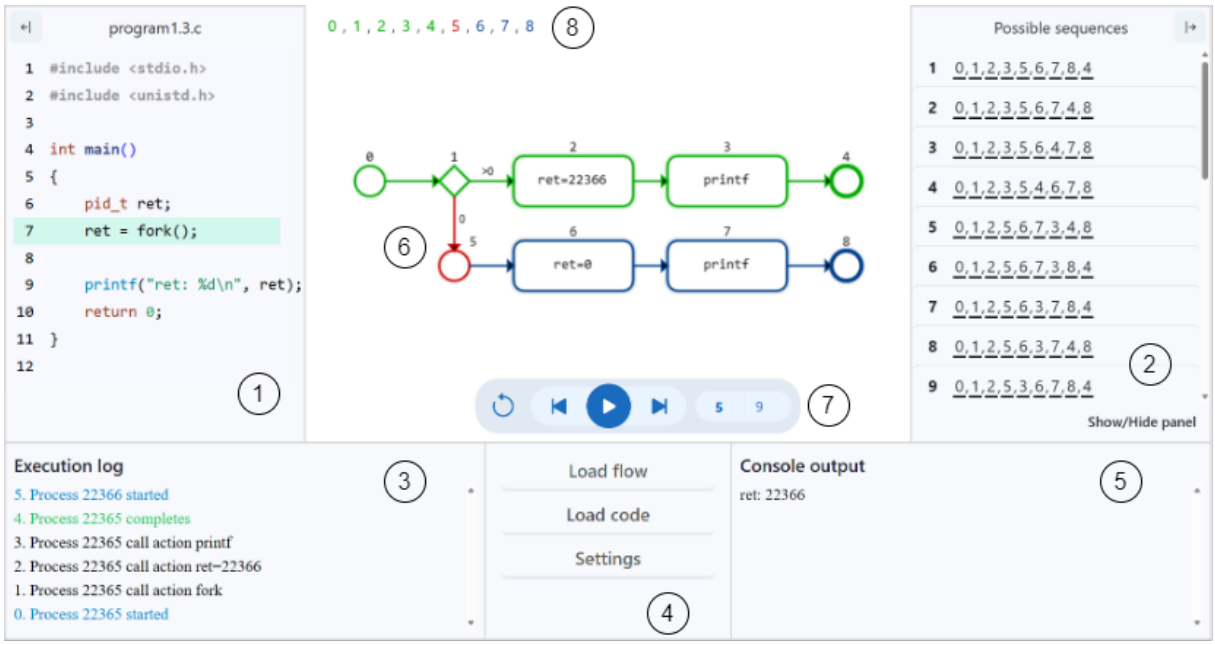


Figure 3.6: Interface web of the application.

- Two lateral panels:
  1. The code panel, displaying the original program code, with the current execution line highlighted;
  2. The possible executions panel presents a selectable list of all potential action sequences the program can execute, assuming identical initial and runtime conditions (e.g., same values for pre-initialized variables and keyboard inputs).
- A control panel with three sections:
  3. The logs section, displaying messages about the actions performed;
  4. The configuration section allows the user to adjust the simulation settings;
  5. The console output section shows the program output.
- A main area, with:
  6. The execution graph that simulated the program;
  7. The simulation controls enable pausing and resuming the simulation, advancing and rewinding to specific actions, looping the execution, and adjusting the

simulation speed;

8. A specific sequence of actions being simulated.

Once all components are synchronized, users can interact with the system using controls. They have the option to adjust the speed of the simulation, allowing observation of the action sequence flow. Additionally, users can modify the action sequence being simulated, facilitating exploration of different execution paths. This interactive experience provides insights into the unpredictable nature of concurrent programs and emphasizes the importance of understanding program execution flow.

The shapes of the graph elements draw inspiration from BPMN diagrams [19], and from the representation used in the IPB classes (Figure 3.4). These are composed of elements representing activities, decisions, and execution flows, which fit the purpose of the visualization. Rectangles depict actions performed, encompassing tasks like variable updates and function calls. Diamonds symbolize fork calls, signifying instances where the program diverges into two processes. Arrows denote the execution flow, depicting the progression from one action to another. Circles mark the initiation and conclusion of process execution, framing the beginning and end points of program flow.

Three different colors were used, to encode the order in time of the various actions: actions yet to be executed (simulated) are represented in blue; those already performed are represented in green; and the current action is represented in red. Note that the diagram is structured in a way that hints at the inherent concurrency/parallelism of the various processes involved: the code paths of a parent process and a direct descendent are represented horizontally side-by-side, to convey this idea. Also, the arrow of time points left to right.

Figure 3.7 shows the various stages involved from the reading of the JSON file to the visualization of the program. These stages First, the JSON file is read. This file follows the format of an array of `ProcessActionFile`. Then it is converted into an asymmetric matrix (an array of arrays) of `ProcessAction`. Each row of this matrix represents the actions of a specific process. This matrix is essential for interpreting the

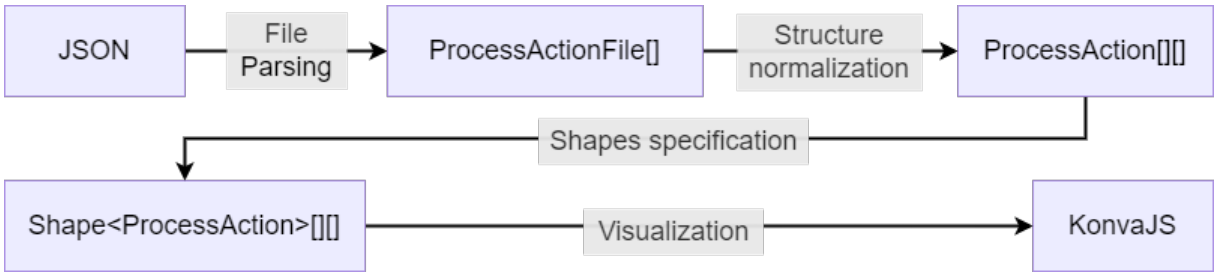


Figure 3.7: Stages for processing the json file, intermediate transformations, and creating the program view.

actions of processes in various operations, such as visualization or the search procedure for possible executions. In the case of visualization, the matrix is converted into an array of `Shape<ProcessAction>`, which contains all the information necessary to create the program visualization using the *Konva* library.

```
type ProcessAction =
  | { id: number; pid: number; line: number; type: 'start'; payload: null }
  | { id: number; pid: number; line: number; type: 'fork'; payload: null; child_pid: number }
  | { id: number; pid: number; line: number; type: 'action'; payload: { ... } }
  | { id: number; pid: number; line: number; type: 'wait'; payload: { ... } }
  | { id: number; pid: number; line: number; type: 'end'; payload: null }
```

Figure 3.8: ProcessAction type.

Figure 3.8 illustrates the `ProcessAction` type, a structure that encapsulates the information and actions necessary to interpret the program:

- A `ProcessAction { type: "start" }` marks the start of a process, derived from a `ProcessActionFile { type: "function_enter", function: "main" }` or `ProcessActionFile { type: "fork_exit", payload: 0 }`.
- A `ProcessAction { type: "fork" }` indicates a fork system call, derived from a `ProcessActionFile { type: "fork_exit", payload: x }`, where x is different from 0.
- A `ProcessAction { type: "end" }` marks the end of a process, derived from a `ProcessActionFile { type: "function_exit", function: "main" }`

or `ProcessActionFile { type: "exit" }`.

- A `ProcessAction { type: "wait" }` indicates a wait or waitpid system call, derived from a `ProcessActionFile { type: "wait" | "waitpid" }`.

- A `ProcessAction { type: "action" }` is a generic representation of an action, derived from a `ProcessActionFile { type: "function_call" | "printf" | "condition" | "variable_assign" | "variable_declare" }`.

It is worth mentioning that `ProcessActionFile { type: "fork_enter" }` does not have a corresponding `ProcessAction`, as its information is not essential for the interpretation of the program. This arrangement of information facilitates the understanding of how the program works and the implementation of operations in the application.

3.2.4 Backend

The backend of a software application is responsible for processing requests from the frontend, performing operations on the Database (DB), and providing the necessary data to the frontend. In *forkSim*, the primary objective of the backend is to manage the processing of C programs submitted by users and to store the execution data for visualization.

This section details the development of the backend, beginning with an explanation of the DB entities, followed by an overview of the architecture and module structure, and concluding with a presentation of the implemented endpoints.

Development was carried out using Python, a high-level, interpreted, and general-purpose programming language. Python was chosen because it was compatible with the libraries used to preprocess the C code. The primary libraries utilized were *FastAPI* and *Databases*. *FastAPI* was used to create the REST API, which handles requests from the frontend. The *Databases* library was employed to connect to the DB, enabling asynchronous execution of SQL queries.

Database

The DBMS used in *forkSim* is *PostgreSQL*, an open-source Relational Database Management System (RDBMS). It was chosen for its robustness, reliability, and support for ACID transactions. The DB stores user data and information about the execution of C programs. The entity-relationship diagram in Figure 3.9 shows the two main entities: *users* and *code_flows*.

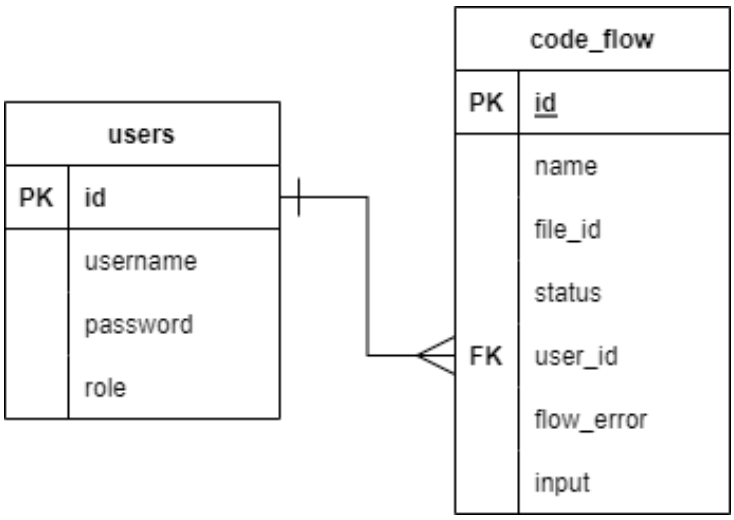


Figure 3.9: Entity-relationship diagram of the database.

The *code_flows* entity is the primary table in the application, responsible for storing information about submitted C programs. It includes the following fields:

- *name*: the name of the program;
- *file_id*: the identifier of the file. Used to name the original C file, the transformed C file, and the JSON execution trace file;
- *status*: the status of the program's execution;
- *user_id*: the identifier of the user who submitted the program;
- *flow_error*: a field for any error generated during execution;
- *input*: a field to store the program's input.

The *users* entity is the table responsible for storing user authentication information. It includes the following fields:

- *username*: the user's name;
- *password*: the user's encrypted password;
- *role*: the user's role in the application, which can be *admin*, *professor*, or *student*.

The *role* field defines the user's permission level in the system. Admins have the ability to perform actions that modify other users' information. The professor role allows the user to control when students can submit code, which is useful for disabling this feature before testing. The student role allows access to existing execution graphs but limits the ability to submit programs only when permitted.

The development of the backend was done incrementally, adding entities and relationships as needed. Initially, the *code_flows* entity was created to store and execute the C programs sent by users. Without the *users* entity, anyone could send C programs to be processed and viewed. At this stage, the main challenge was to integrate the API with the C program execution service. Then, the authentication system was implemented, and the *users* entity was created to ensure the security and privacy of users. Finally, the *users* entity was associated with the *code_flows* entity to identify the owner of each submitted C program, allowing users to update and delete their programs.

Structure and Operations

The backend of the *forkSim* project was developed following a layered software architecture pattern to separate the responsibilities of each module. Figure 3.10a shows the directory tree, with colors representing the layer of each module. There are three main layers: control (green), logic (yellow), and data (red). The control layer is the highest, responsible for defining the HTTP communication interface, configuring endpoint permission levels, and invoking functions from the logic layer (*main.py* and *controllers/**).

The other modules are responsible for the configuration of the application, containing:

(a) Directory tree of the API. (b) Module Communication Graph of the API.

Figure 3.10: Backend structure and operation graph.

- *env.py*: the definition of environment variables.
- *exceptions.py*: the definition of exceptions.
- *mappers.py*: the definition of data mappings.
- *models.py*: the definition of models and Data Transfer Objects (DTOs).
- *resources.py*: the definition of paths for storing files.
- *connection.py*: the configuration of the database connection.
- *init_database.py*: the initialization of the database.
- *process_code_flow_job.py*: the definition of the queue for executing C processes.

To understand the backend operation, a simplified Module Communication Graph (MCG) is presented in Figure 3.10b. The *main.py* module is the entry point of the application, responsible for configuring the FastAPI application, defining global endpoints, and registering all control modules. The control modules access the service modules, which in turn use the data modules and others to perform the operations required by the client.

The code in Listing 3.3 shows the implementation of the *code_flow* controller, responsible for managing the endpoints related to the C programs. The code sets up an API router with a specific prefix and tags. This router controls access to various operations related to code and flow files. To ensure proper authorization, a function is called to define the user role required to access these endpoints. Following this, a series of asynchronous endpoint functions are defined. Each function is decorated to handle a specific HTTP method (e.g., GET, POST, etc.) and provides a description of its operation.

The code in Listing 3.4 shows the implementation of the *code_flow* service, responsible for managing the operations related to the C programs. The code defines a *CodeFlowService* class, which is initialized with a *code_flow_repository* and a *process_code_flow_job*. The class includes asynchronous functions for showing, listing, updating, and deleting code and flow files. Additionally, a *get_code_flow_service* function is defined to create and return an instance of *CodeFlowService*. This function uses dependency injection to obtain instances of *CodeFlowRepository* and *ProcessCodeFlowJob*, which are then passed

to the *CodeFlowService* constructor. This set up ensures that the service has access to the necessary resources and logic for handling code flow operations.

```
router = APIRouter(prefix="/code-flow", tags=["CodeFlow"])
get_token_with_router_roles = get_token_with_role(
  UserRole.ADMIN, UserRole.PROFESSOR)

@router.get("/{id}/", description="Show code and flow files")
async def code_flow_show(
    id: int,
    service: CodeFlowService = Depends(get_code_flow_service),
    token: TokenData = Depends(get_required_token),
) → CodeFlowShow:
    return await service.code_flow_show(id, token.user)

@router.get("/", description="List code and flow files")
async def code_flow_index(...) → List[CodeFlowShow]:
    return await service.code_flow_index(...)

@router.post("/", description="Store code and flow files")
async def code_flow_store(...) → CodeFlowShow:
    return await use_case.execute(...)

@router.put("/{id}/", description="Update code and flow files")
async def code_flow_update(...) → CodeFlowShow:
    return await service.code_flow_update(...)

@router.delete("/{id}/", description="Delete code and flow files")
async def code_flow_delete(...) → None:
    return await service.code_flow_delete(...)
```

Listing 3.3: code_flow controller

This API exposes the following endpoints:

- **GET /docs**: Access to the Swagger documentation of the API.
- **GET /redoc**: Access to the ReDoc documentation of the API.

```python
class CodeFlowService:
    def __init__(
        self,
        code_flow_repository: CodeFlowRepository,
        process_code_flow_job: ProcessCodeFlowJob
    ) -> None:
        self.code_flow_repository = code_flow_repository
        self.process_code_flow_job = process_code_flow_job

    async def code_flow_show(self, ...) -> CodeFlowShow:
        ...

    async def code_flow_index(self, ...) -> List[CodeFlowShow]:
        ...

    async def code_flow_update(self, ...) -> CodeFlowShow:
        ...

    async def code_flow_delete(self, ...) -> None:
        ...

def get_code_flow_service(
    code_flow_repository: CodeFlowRepository =
      Depends(get_code_flow_repository),
    process_code_flow_job: ProcessCodeFlowJob =
      Depends(get_process_code_flow_job),
) -> CodeFlowService:
    return CodeFlowService(
      code_flow_repository, process_code_flow_job)
```

Listing 3.4: code_flow service

- **GET /static/files/{file_path}**: Access to static files, such as C code files and execution trace files.

- **POST /api/auth/login**: Authenticate the user, responding with an authentication token.

- **POST /api/auth/refresh**: Refresh the authentication token, responding with a new token.
- **POST /api/auth/change-password**: Change the password of the authenticated user.
- **POST /api/auth/is-logged-in**: Validate the user's token indicating whether they are authenticated.
- **POST /api/user**: Create a new user; performed only by *admin* users.
- **PUT /api/user**: Update the user's data; performed only by *admin* users.
- **GET /api/user/me**: Get the data of the authenticated user, responding with the user's data.
- **GET /api/code-flow**: Get the public and private code flows.
- **GET /api/code-flow/{id}**: Get a specific code flow by its identifier.
- **POST /api/code-flow**: Create a new code flow from a C file.
- **PUT /api/code-flow/{id}**: Update a specific code flow by its identifier; performed only by the user who created the flow.
- **DELETE /api/code-flow/{id}**: Delete a specific code flow by its identifier; performed only by the user who created the flow.

3.2.5 Application Architecture and Containerization

The architecture of the *forkSim* plataform defines the interconnection between the frontend and backend applications, the database, and the C program execution service. All four components run in separate Docker containers. The configuration of the application is managed using *Docker Compose*, a tool designed to define and run multi-container applications with Docker.

Figure 3.11 presents the high-level architecture of the *forkSim* platform. The frontend application, built with TS and React, is accessed by a web client on port 8080. The

frontend communicates with the backend application, built with FastAPI, on port 8000. The backend has access to the PostgreSQL database on port 5432 and the C program execution service on port 8001, which is also a FastAPI application.

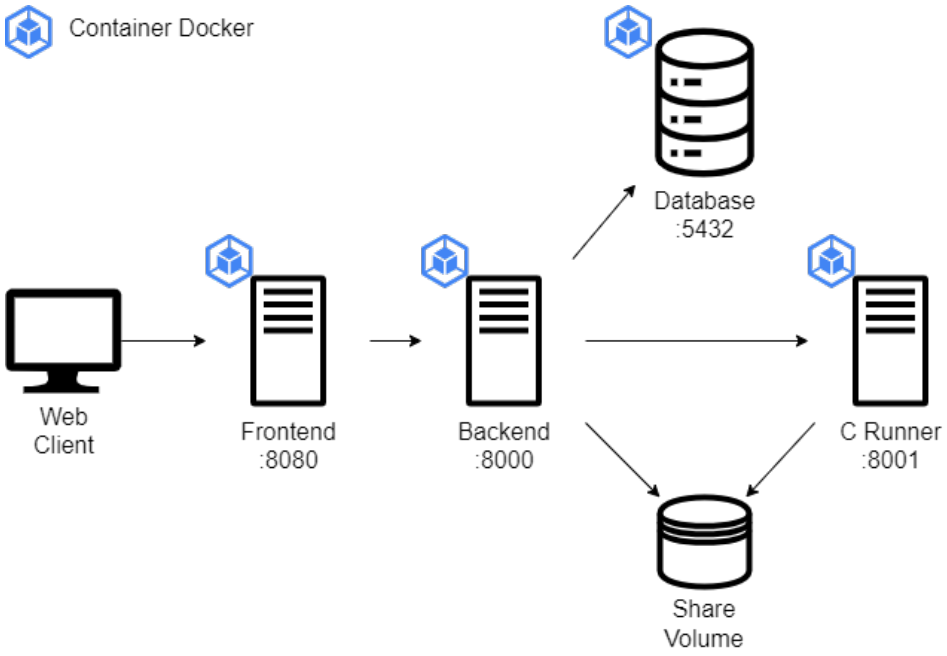


Figure 3.11: Architecture diagram of the application.

The C program execution service was separated from the main backend to ensure the scalability, maintainability, and security of the application. This service is equipped with development tools for the C language to compile and execute the programs sent by users. To facilitate the implementation of this service, the C files are accessed from a shared volume with the backend.

3.3 Final Considerations

This chapter described the creation of the *forkSim* application, a tool for visualizing the execution of C programs with fork system calls. It covers the requirements, design, implementation, and evaluation of the application. The following chapter presents the findings of this work, demonstrating a use case for the application through the entire

process of transforming, executing, and visualizing a C program. The findings from classroom research and the usability questionnaire were also presented.

Chapter 4

Use Case and Evaluation

This chapter covers the utilization of *forkSim* under two perspectives. First (internal view) it focus on the various stages which a specific code example goes through. Second (external view) it describes a real-word evaluation (methodology and results) conducted with students of SO classes at IPB.

4.1 Usage Example

The code in Listing 4.1 is an example of C code that can be used in the *forkSim* application. It includes all system calls currently supported by *forkSim* (*fork*, *wait*, and *waitpid*).

To generate the execution trace, this code goes through the transformation process generating the code presented in Listing 4.2. During processing, the inspector library header is added to the program. The *inspector_function_enter()* and *inspector_function_exit()* functions are added to the beginning and end of each function, respectively. Some functions are intercepted literally, for example *printf()* and *fork()*, in the *inspector_[function name]()* pattern. Other functions are simply marked with *inspector_function_call()*. In conditionals and loops, the conditional expression is passed literally and as a string to the *inspector_condition()* function, allowing the processing of the expression and its result.

After authenticating, sending the C code and executing the program, the user can view the generated execution trace. The trace is presented in JSON format, as shown in

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
  pid_t ret;
  ret = fork();
  if (ret == 0) {
    exit(0);
  }
  waitpid(ret, NULL, 0);
  printf("PARENT %d of the CHILD %d\n", getpid(), ret);
  getchar();
  return 0;
}
```

Listing 4.1: C code example

Figure 4.1. The graph generated by the visualization tool is shown in Figure 4.2. The graph shows the main process branching into two child processes after the fork call. The main process waits for the child process to finish, and then prints a message. The child process is terminated after the exit call, without performing any other operations.

A teacher can use examples like this and show to the students its visualization and explore all possible execution combinations. Students can use the tool to compare the *process tree* generated by their own programs or those from the textbook. For that, the user just have to upload the program and run it; the rest will be automatically computed.

4.2 Evaluation

The intention of the evaluation of the *forkSim* application was to assess its usability and effectiveness in the classroom. Questions about the interest of students about the subject, the understanding of the content, and the application's contribution to the learning process were addressed. Section 4.2.1 describe the research conducted in the classroom,

```c
#define INSPECTOR_IMPLEMENTATION
#include <inspector.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
  inspector_function_enter("main", 4);
  pid_t ret;
  inspector_variable_declare("main", 5, "pid_t", "ret");
  ret = inspector_fork("main", 6);
  inspector_variable_assign("main", 6, "pid_t", "ret", &ret);
  if (inspector_condition(
        "main", 7, "if", ret == 0, "ret == 0")) {
    inspector_exit("main", 8, 0);
  }
  inspector_waitpid("main", 10, ret, NULL, 0);
  inspector_printf("main", 11, "PARENT %d of the CHILD %d\n",
    (inspector_function_call("main", 11, "getpid"), getpid()),
      ret);
  (inspector_function_call("main", 12, "getchar"), getchar());
  inspector_function_exit("main", 14);
  return 0;
}
```

Listing 4.2: Transformed C code example

and the usability questionnaire applied to the students in Section 4.2.2.

4.2.1 Classroom Evaluation

Methodology of Classroom Evaluation

The research in the classroom was an experiment conducted by the teachers of the Operating Systems discipline to evaluate the effectiveness of the *forkSim* application. 5 classes of students were selected to participate in the experiment, totaling 79 students. 3 classes (35 students) used the application during the practical classes, and the other 2 classes (44

```
[
  { "time": 1713890812029992, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 4,
    "type": "function_enter", "payload": "" },
  { "time": 1713890812030002, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 5,
    "type": "variable_declare", "payload": { "type": "pid_t", "name": "ret" } },
  { "time": 1713890812030006, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 6,
    "type": "fork_enter", "payload": "" },
  { "time": 1713890812030073, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 6,
    "type": "fork_exit", "payload": 29 },
  { "time": 1713890812030082, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 6,
    "type": "variable_assign", "payload": { "type": "pid_t", "name": "ret", "value": 29 } },
  { "time": 1713890812030085, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 7,
    "type": "condition", "payload": { "cause": "if", "value": 0, "expression": "ret == 0" } },
  { "time": 1713890812030245, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 10,
    "type": "waitpid", "payload": { "pid": 29, "status": 0 } },
  { "time": 1713890812030254, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 11,
    "type": "printf", "payload": "PARENT 28 of the CHILD 29\n" },
  { "time": 1713890812030285, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 11,
    "type": "function_call", "payload": { "function": "getpid" } },
  { "time": 1713890812030292, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 12,
    "type": "function_call", "payload": { "function": "getchar" } },
  { "time": 1713890812030294, "depth": 0, "pid": 28, "parent_pid": 22, "function": "main", "line": 14,
    "type": "function_exit", "payload": "" },
  { "time": 1713890812030167, "depth": 1, "pid": 29, "parent_pid": 28, "function": "main", "line": 6,
    "type": "fork_exit", "payload": 0 },
  { "time": 1713890812030179, "depth": 1, "pid": 29, "parent_pid": 28, "function": "main", "line": 6,
    "type": "variable_assign", "payload": { "type": "pid_t", "name": "ret", "value": 0 } },
  { "time": 1713890812030183, "depth": 1, "pid": 29, "parent_pid": 28, "function": "main", "line": 7,
    "type": "condition", "payload": { "cause": "if", "value": 1, "expression": "ret == 0" } },
  { "time": 1713890812030186, "depth": 1, "pid": 29, "parent_pid": 28, "function": "main", "line": 8,
    "type": "exit", "payload": "" }
]
```


Figure 4.1: Execution trace in JSON format of the example

students) did not use the application. After the end of this content, a practical exercise was applied for the students to draw the execution graph of a C program. In order to compare the scores of the students who used the application with those who did not, the exercise had a maximum grade of 20 points.

The research lasted approximately 1 month to be completed. In the first week, the

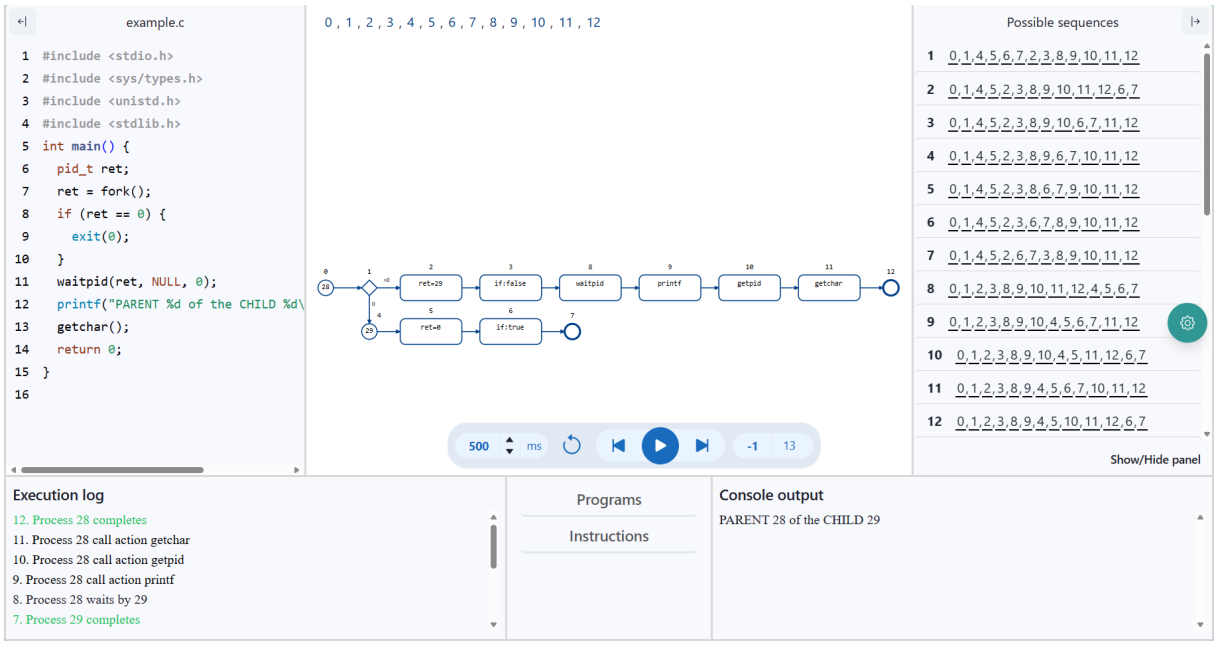


Figure 4.2: Visualization of the execution graph of the example

application was presented to the students who would use it. In the following week, all students did the evaluative activity to draw the execution graph of 2 C programs, presented in Listings 4.3 and 4.4. In the following weeks, this program was loaded into the application so that the teacher could present the correct execution graph.

Listings 4.3 and 4.4 present two very similar programs. The first program (Listing 4.3) presents an *if* with the condition *fork() > 0*, while the second program (Listing 4.4) presents an *if* with the condition *fork() == 0*. These conditions are sufficient for the programs to have different behaviors.

Comparing the execution graphs of both programs, it is possible to notice their differences. In Program A (Listing 4.3), the initial process creates all child processes, as shown in the graph in Figure 4.3a. In Program B (Listing 4.4), each process created, the parent process is terminated, and the child process continues execution by creating new child processes, as shown in the graph in Figure 4.3b. In both graphs, the values of the variables i and j are another important detail to understand the program's behavior.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int i, j;
    i = 1;
    j = 1;
    do
    {
        if (fork() > 0) // (1)
        {
            j += 1;
        }
        else
        {
            j += 2;
        }
        i++;
    } while (i < 2 || j < 3);
    return 0;
}
```

Listing 4.3: Program A

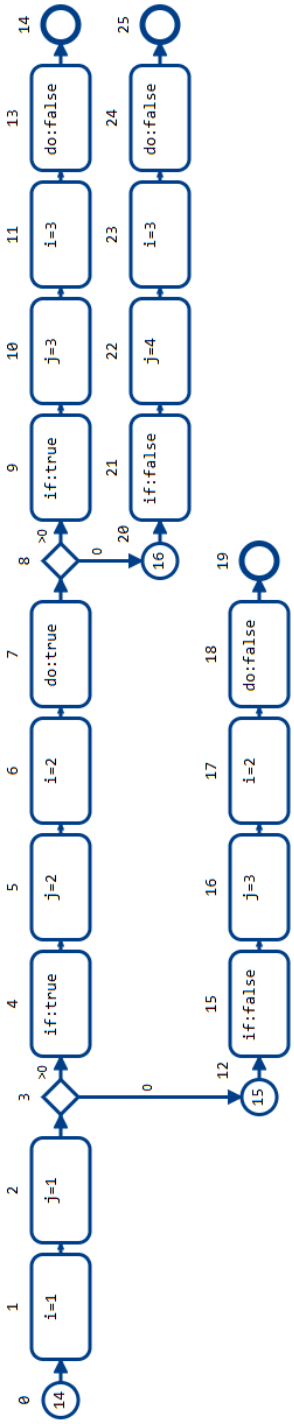
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int i, j;
    i = 1;
    j = 1;
    do
    {
        if (fork() == 0) // (2)
        {
            j += 1;
        }
        else
        {
            j += 2;
        }
        i++;
    } while (i < 2 || j < 3);
    return 0;
}
```

Listing 4.4: Program B

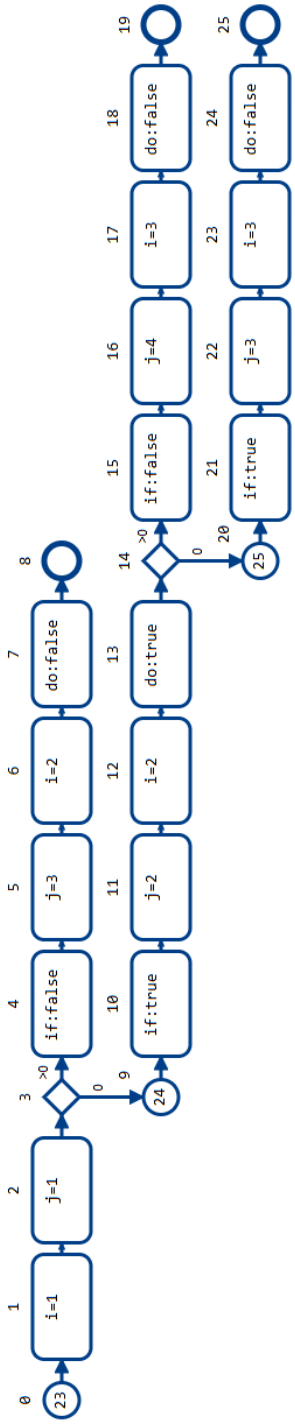
Results of Classroom Evaluation

The classroom evaluation conducted with 79 students from 5 different groups showed a better result for the students who used the tool. Groups A, B and D that used and the groups C and E that did not use the tool had a mean score of 10.86 and 8.13, respectively. Table 4.1 presents metrics for each group of students, derived from Table 4.2 data.

The graph 4.4a shows a predominance of male students, with 68 boys (86%) and 11 girls (14%). It is possible to observe that of the students who did not use the tool, 43 (98%) are boys and only 1 (2%) is a girl. Of the students who used the tool, 25 (71%) are boys and 10 (29%) are girls.



(a) Program A



(b) Program B

Figure 4.3: Execution graphs of the evaluation programs.

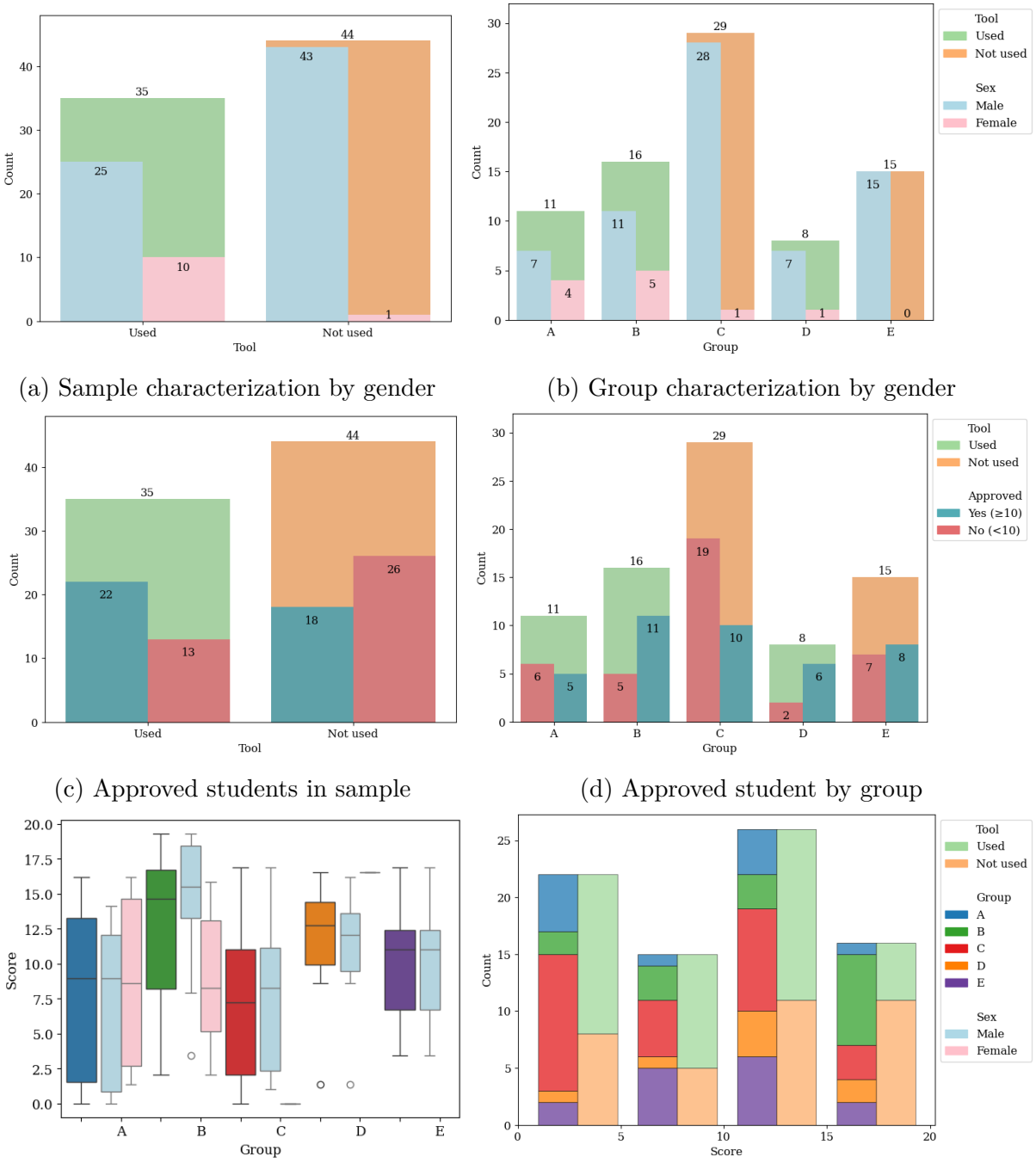
Table 4.1: Evaluation summary

Group	Count	Mean	Median	SD	Min.	Max.
A	11	7.62	8.97	6.41	0	16.21
B	16	12.74	14.655	5.66	2.07	19.31
C	29	7.23	7.24	5.2	0	16.9
D	8	11.55	12.76	4.91	1.38	16.55
E	15	9.86	11.03	4.16	3.45	16.9
General	79	9.34	10	5.6	0	19.31

Table 4.2: Scores of the evaluation exercise

Group A		Group B		Group C		Group D		Group E	
Score	Gender	Score	Gender	Score	Gender	Score	Gender	Score	Gender
8.97	M	8.28	F	16.21	M	13.45	M	6.9	M
0.0	M	12.07	M	13.1	M	16.21	M	11.38	M
16.21	F	5.17	F	7.24	M	1.38	M	12.76	M
1.72	M	14.48	M	13.1	M	13.79	M	15.86	M
3.1	F	2.07	F	11.38	M	16.55	F	3.45	M
0.0	M	18.62	M	2.07	M	12.07	M	4.83	M
1.38	F	18.28	M	9.66	M	8.62	M	3.45	M
11.72	M	16.21	M	10.0	M	10.34	M	9.31	M
14.14	F	14.83	M	11.03	M	-	-	12.76	M
12.41	M	7.93	M	1.38	M	-	-	12.07	M
14.14	M	18.62	M	0.0	F	-	-	8.97	M
-	-	15.52	M	14.83	M	-	-	16.9	M
-	-	13.1	F	1.03	M	-	-	11.72	M
-	-	3.45	M	5.17	M	-	-	6.55	M
-	-	15.86	F	10.0	M	-	-	11.03	M
-	-	19.31	M	12.76	M	-	-	-	-
-	-	-	-	1.38	M	-	-	-	-
-	-	-	-	7.24	M	-	-	-	-
-	-	-	-	16.9	M	-	-	-	-
-	-	-	-	1.38	M	-	-	-	-
-	-	-	-	2.41	M	-	-	-	-
-	-	-	-	9.31	M	-	-	-	-
-	-	-	-	3.45	M	-	-	-	-
-	-	-	-	3.79	M	-	-	-	-
-	-	-	-	1.72	M	-	-	-	-
-	-	-	-	9.66	M	-	-	-	-
-	-	-	-	1.72	M	-	-	-	-
-	-	-	-	2.41	M	-	-	-	-
-	-	-	-	9.31	M	-	-	-	-

The number of students per group and gender can be observed in the graph 4.4b.



(a) Sample characterization by gender

(b) Group characterization by gender

(c) Approved students in sample

(d) Approved student by group

(e) Box Plot of the Scores by tool use, by group and by gender

(f) Stacked Histogram of the Scores by tool use and by group

Figure 4.4: Charts about the evaluation exercise

Groups A and B have the highest number of girls, with 4 and 5 respectively. Groups C and D have 1 girl each, while group E has no girls.

The graphs 4.4c and 4.4d show the relationship between approved and not-approved students. Of all students, 40 (51%) received a score above the mean, while 39 (49%) received a grade below the mean. Of the students who used the tool, the proportion of approved students is higher, with 22 (63%) approved and 13 (37%) not-approved. Of the students who did not use the tool, 18 (41%) were approved and 26 (59%) were not-approved. Group C had the highest number of not-approved students, with 19 (66%) not-approved and 10 (34%) approved. Group B had the highest number of approved students, with 11 (69%) approved and 5 (31%) not-approved.

The graphs 4.4e and 4.4f show the distribution of student scores from the box plot and stacked histogram. The box plot of group B shows that boys performed better than girls. For group A, girls performed slightly better than boys. The only girl in group D had the highest score in the group.

The stacked histogram shows that most students scored between 10 and 15. Group C has the highest number of students with a score below 5, while group B has the highest number of students with a score above 15. The scores of the evaluation and the gender of each student in each group are available in Table 4.1.

4.2.2 Usability Questionnaire

Methodology of Usability Questionary

A questionnaire was developed with questions about the user experience with the *forkSim* application to evaluate the usability of the application. The questionnaire was applied to the students of the Operating Systems discipline, who used the application during the practical classes. 15 questions were asked, divided into 6 sections: previous use, exercise given in class, components of the tool, keyboard shortcuts, and an open question in the end. The scale used in the response to some questions represents a scale from 1 to 5, where 1 represents the lowest degree of agreement and 5 the highest degree of agreement.

The questions of the questionnaire were as follows:

- Previous Use

1. Have you used the tool before this class? Yes / No
2. Did you make more effort to perform the proposed exercises in class? 1-5
3. Did you feel more self-confident in solving the exercises? 1-5
4. Did you feel more autonomy in learning? 1-5

- About the Exercise in Class

5. How many processes were created, besides the initial process? (numeric)
6. What were the final values of the variables i and j in each process? Identify each process based on its PID. (open)

- Use of the Tool in Class

7. Did you find it easier to interpret the program's operation? 1-5
8. Did you feel more motivated to learn on your own? 1-5

- Components of the Tool

9. How useful is the panel with the code being executed? 1-5
10. How useful is the panel with the possible sequences? 1-5
11. How useful is the panel with the actions being executed? 1-5
12. How useful is the panel with the output of the program's console prints (printf)? 1-5

- Keyboard Shortcuts

13. How often did you use keyboard shortcuts? 1-5
14. How familiar and usual were the shortcuts? 1-5

- Open

15. Give suggestions and criticisms about the application. Talk about bugs and errors you have experienced, actions you felt needed to exist, and actions that hindered you. (open)

The aim of each set of questions put forward was the following:

- Previous Use: understand the level of experience of the students with the application;
- Exercise in Class: evaluate the performance of the students in solving the proposed exercises, related to the program presented in the Listing 4.5 and the execution graph presented in Figure 4.5;
- Use of the Tool in Class: evaluate the experience of the students with the application during the class;
- Components of the Tool: evaluate the usefulness of the application components;
- Keyboard Shortcuts: evaluate the usability of the keyboard shortcuts of the application;
- Open question: collect suggestions and criticisms about the application.

For question 5, the correct answer is 4 processes, in addition to the initial process. Question 6 is open, and the correct answer is, with the processes identifiers of Figure 4.5:

- Process 14: i = 2, j = 16
- Process 15: i = 2, j = 0
- Process 16: i = 3, j = 0
- Process 17: i = 2, j = 0
- Process 17: i = 3, j = 0

```c
#include <unistd.h>
int main()
{
    int i, j;
    j = -1;
    i = 0;
    do
    {
        i++;
        j = fork();
        if (j == 0)
        {
            i++;
            fork();
        }
    } while (i < 2);
    return 0;
}
```

Listing 4.5: Program for the questionnaire activity.

Results of Usability Questionnaire

The usability questionnaire was available for a period of 1 month and received 15 responses. Despite the small number of responses, it was possible to obtain relevant information about the usability of the tool.

Among the 15 questions, 11 used the Likert scale from 1 to 5, where 1 is totally disagree and 5 is totally agree. Figure 4.6 shows a heat map of the responses to these questions. All questions had an average of 3 or higher, indicating that the responses were neutral or positive.

Questions 2, 3, and 4, about the effort to learn the content, self-confidence to solve the exercises, and the feeling of autonomy had averages of 3, 3.5, and 3.42. This indicates that students were not sure about the questions or felt neutral. Question 2 had 1 response totally disagreeing, showing a case of dissatisfaction. Questions 7 and 8, about the use of

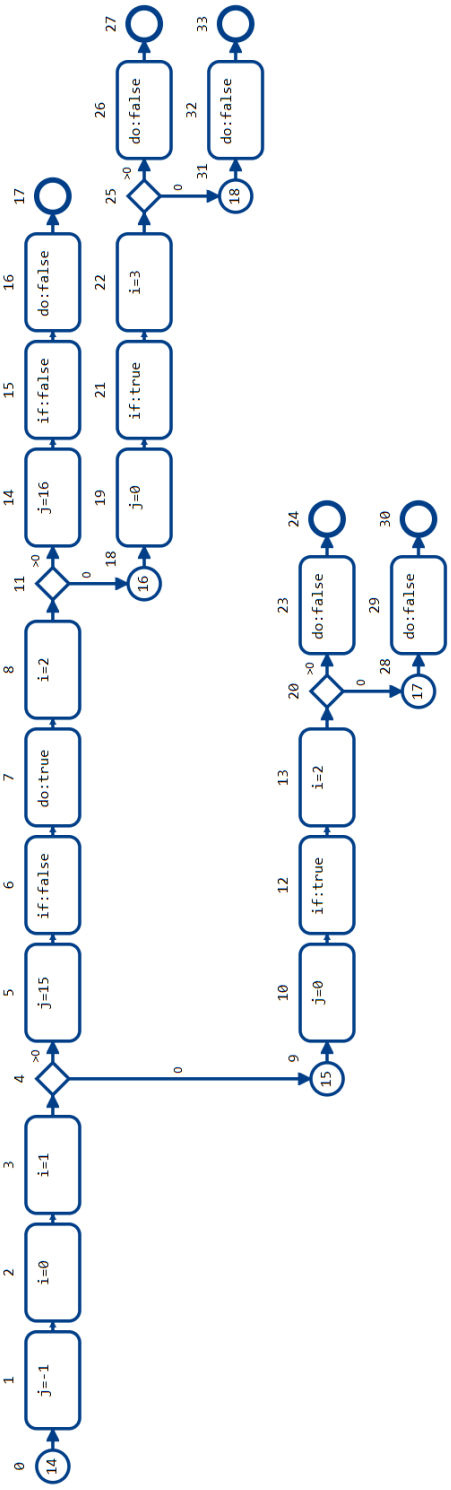


Figure 4.5: Execution graph of the program for the questionnaire activity.

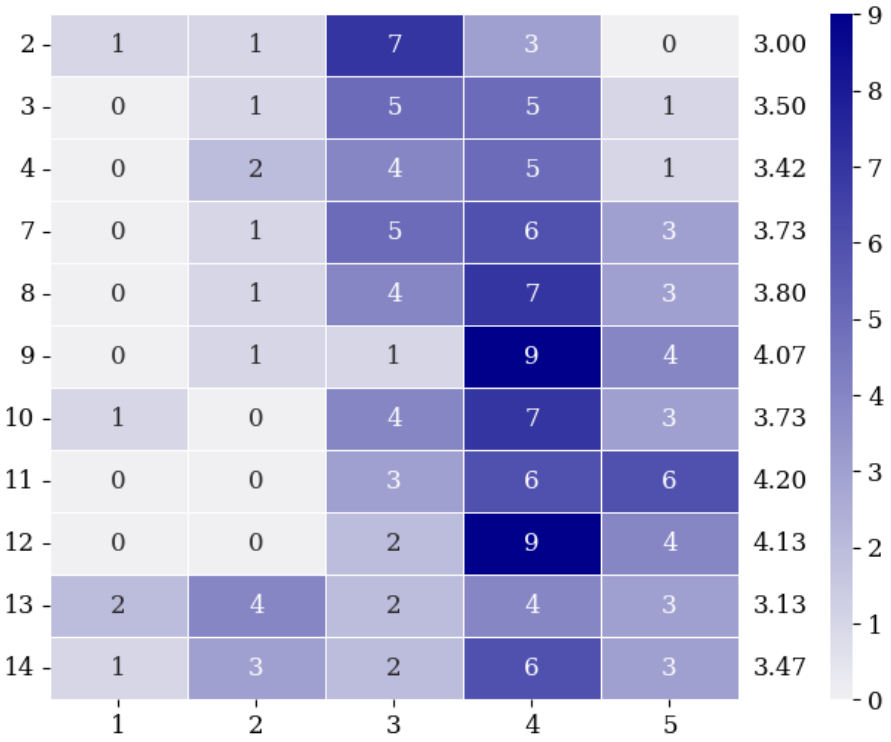


Figure 4.6: Heatmap of usability questionnaire responses.

the tool to understand the content had averages of 3.73 and 3.8, with 3 responses totally agreeing in both.

The questions 9, 10, 11, and 12 about the importance of the components had 3 averages above 4, indicating a strong agreement. About keyboard shortcuts, questions 13 and 14 had averages of 3.13 and 3.47, with responses in all values, indicating strong agreement by some students and disagreement by others. This is due to the habit of using keyboard shortcuts, which can vary from student to student.

Questions 5 and 6 about the activity using the tool showed a negative performance. Question 5 had 5 correct answers (33%). Question 6 had only 2 correct answers (13%), and 4 answers appropriate for correction (26%).

Finally, the optional and open question 15 had 3 responses. The answers included praise for the tool and the following suggestions:

- Allow the execution of C programs sent by students;

- Make the tool usable on mobile devices;
- Offer an offline version of the tool.

4.3 Final Considerations

This chapter presented a use case and a classroom evaluation of *forkSim*. The use case demonstrated how the application generates an execution trace from C code. The evaluation provided information on the performance of students who used the tool compared to those who did not. Additionally, the usability questionnaire revealed students' perceptions of the tool. The first suggestion was implemented during the development of the platform. The others are still considered relevant, but were left for future work.

Chapter 5

Conclusion

Concurrent programming represents a significant paradigm shift for many students, making the learning process challenging due to its unpredictable nature, non-deterministic behavior, and control flow complexities inherent in Unix-like systems. This thesis addresses these challenges through the development of *forkSim*, an application for the interactive visualization of execution graphs, designed to facilitate a better understanding of concurrent programming concepts. The work involved the development of several components:

1. An inspection library, which records the execution actions of the program;
2. A preprocessing tool that injects inspection functions into the program's source code;
3. A web application for visualizing the execution graph, interfacing with the server to obtain this data;
4. A server that manages the execution of programs and facilitates communication with the web application.

forkSim, currently live at `http://forksim.estig.ipb.pt:8080/`, provides users the ability to visualize the original sequence of program execution steps or simulate arbitrary step sequences. It also offers the capability to add new program examples to generate corresponding execution graphs, thus enhancing its utility as an educational resource.

The evaluation of the tool through a usability questionnaire revealed that users found it intuitive and easy to use. Additionally, the evaluation activity showed that students who utilized the tool performed better in understanding concurrent programming concepts compared to those who did not use it. These findings highlight the tool's effectiveness in supporting the learning process.

Overall, the tool achieved its primary objective of generating interactive visualizations of concurrent programs from the source code. The survey conducted with students confirmed its effectiveness as a learning aid. However, the tool currently has some limitations. It supports only the C language and requires knowledge of Unix-like systems for creating examples. Also, to avoid overloading the graph, it is expected to handle only simple programs.

Future work could focus on extending the tool's capabilities to support additional programming languages and operating systems. Enhancements could also include handling more complex programs and incorporating other forms of concurrent programming, such as threads and message passing. By addressing these limitations, the tool could become a more versatile and comprehensive resource for learning concurrent programming and also support mobile platform and offline usage, as suggested by some users.

Finally, it should be mentioned that a scientific paper based on this work was accepted in the SLATE/2024 conference [49], a recognition of the merits of this research.

Bibliography

[1] L. Almeida Mapurunga and E. B. Elcyana Bezerra Carvalho, "A memória de longo prazo e a análise sobre sua função no processo de aprendizagem," *Revista de Ensino, Educação e Ciências Humanas*, vol. 19, no. 1, pp. 66–72, 2018. DOI: 10.17921/2447-8733.2018v19n1p66-72. [Online]. Available: https://revistaensinoeeducacao.pgsscogna.com.br/ensino/article/view/4443.

[2] F. A. Carvalho, F. F. Borges, G. F. Silva, T. O. Borges, and E. L. Bispo Jr., "Ensino de provas por indução em grafos utilizando uma ferramenta visual de algoritmos," in *Conferência sobre Educação em Computação*, Universidade Federal de Goiás, Brasil, 2017.

[3] H. F. Fritsch, E. Silva, J. Souza, *et al.*, "Ordena–um jogo educacional para auxílio ao ensino de métodos de ordenação," in *ENCOINFO–Congresso de Computação e Tecnologias da Informação*, vol. 18, 2016, pp. 132–131.

[4] T. D. N. d. Sousa *et al.*, "Um interpretador gráfico de comandos baseado na jvm como ferramenta de ensino de programação, algoritmos e estruturas de dados," 2013.

[5] G. Project, *Gdb: The gnu project debugger*, https://sourceware.org/gdb/, Accessed: 2024-02-04, 2024.

[6] F. S. Foundation, *Gdb front ends and other tools*, https://sourceware.org/gdb/wiki/GDBFrontEnds/, Accessed: 2024-05-21, 2023.

[7] M. Ben-Ari, *Principles of concurrent and distributed programming*. Addison-Wesley, 2006, ISBN: 9780321312839.

[8] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2006.

[9] A. Tanenbaum, *Modern Operating Systems*. Prentice Hall, 2014.

[10] M. Kerrisk, *Fork(2) - linux manual page*, `https://www.man7.org/linux/man-pages/man2/fork.2.html`, Accessed: 2024-05-28, 2024.

[11] M. Kerrisk, *Wait(2) — linux manual page*, `https://man7.org/linux/man-pages/man2/wait.2.html`, Accessed: 2024-06-15, 2023.

[12] U. Manber, *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Professional, 1999.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006, ISBN: 9780321486813.

[14] B. Cabral, "Instrumentação de código na plataforma .net," Ph.D. dissertation, Jun. 2005.

[15] G. Kiczales, J. Lamping, A. Mendhekar, *et al.*, "Aspect-oriented programming," in *ECOOP'97. Proceedings of the 11th European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science (LNCS), vol. 1241, 1997, pp. 220–242, ISBN: 3-540-63089-9. DOI: `10.1007/BFb0053381`.

[16] T. Lindholm and F. Yellin, *Compiling for the Java Virtual Machine*. Addison-Wesley, 1999, ISBN: 978-0201432947.

[17] A. Bastidas, M. Pérez, and J. Meza, "Transpilers: A systematic mapping review of their usage in research and industry," *Applied Sciences*, vol. 13, no. 6, p. 3667, 2023. DOI: `10.3390/app13063667`.

[18] G. Project, *Gcc, the gnu compiler collection*, `https://gcc.gnu.org/`, Accessed: 2024-05-28, 2024.

[19] O. M. Group, *Business process model and notation (bpmn)*, `https://www.omg.org/spec/BPMN/2.0/`, Accessed: 2024-05-28, 2024.

[20] Microsoft, *Typescript language*, https://www.typescriptlang.org/, Accessed: 2024-06-15, 2024.

[21] M. O. Source, *React js*, https://reactjs.org/, Accessed: 2024-05-14, 2024.

[22] Vercel, *Chakra ui*, https://chakra-ui.com/, Accessed: 2024-05-14, 2024.

[23] KonvaJS, *Konva website*, https://konvajs.org, Accessed: 2024-05-13, 2024.

[24] J. J. ". Sarjeant, *Axios*, https://axios-http.com/, Accessed: 2024-05-14, 2024.

[25] P. S. Foundation, *Python*, https://www.python.org/, Accessed: 2024-05-14, 2024.

[26] tiangolo, *Fastapi*, https://fastapi.tiangolo.com/, Accessed: 2024-05-14, 2024.

[27] T. Christie, *Databases*, https://pypi.org/project/databases/, Accessed: 2024-05-14, 2024.

[28] Pydantic, *Pydantic*, https://pydantic-docs.helpmanual.io/, Accessed: 2024-05-14, 2024.

[29] Uvicorn, *Uvicorn*, https://www.uvicorn.org/, Accessed: 2024-05-14, 2024.

[30] Postgres, *Postgres*, https://www.postgresql.org/, Accessed: 2024-05-14, 2024.

[31] D. Beazley, *Ply (python lex-yacc)*, https://ply.readthedocs.io/en/latest/ply.html, Accessed: 2024-05-14, 2024.

[32] M. Brunsfeld, *Tree-sitter*, https://github.com/tree-sitter/tree-sitter, Accessed: 2024-05-14, 2024.

[33] Docker, *Docker*, https://www.docker.com/, Accessed: 2024-05-14, 2024.

[34] Docker, *Docker compose*, https://docs.docker.com/compose/, Accessed: 2024-05-14, 2024.

[35] Figma, *Figma*, https://www.figma.com/, Accessed: 2024-05-14, 2024.

[36] Graphviz, *Graphviz website*, https://graphviz.org, Accessed: 2024-05-13, 2023.

[37] Observable, *D3.js website*, https://d3js.org/, Accessed: 2024-05-13, 2024.

[38] W3C, *What is svg?* https://www.w3.org/Graphics/SVG/, Accessed: 2024-05-13, 2024.

[39] E. Shinan, *Lark*, https://github.com/lark-parser/lark, Accessed: 2024-05-14, 2024.

[40] T. Parr, *Antlr*, https://www.antlr.org/, Accessed: 2024-05-14, 2024.

[41] P. J. Guo, "Online python tutor: Embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13, Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 579–584, ISBN: 9781450318686. DOI: 10.1145/2445196.2445368. [Online]. Available: https://doi.org/10.1145/2445196.2445368.

[42] P. Guo, "Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1235–1251, ISBN: 9781450386357. DOI: 10.1145/3472749.3474819. [Online]. Available: https://doi.org/10.1145/3472749.3474819.

[43] P. J. Guo, *Python tutor*, https://pythontutor.com/, Accessed: 2024-02-19.

[44] C. Balsa, L. M. Alves, M. J. Pereira, P. Rodrigues, and R. Lopes, "Graphical simulation of numerical algorithms : An approach based on code instrumentation and java technologies," *CSEDU 2012 - 4th International Conference on Computer Supported Education*, 2012. [Online]. Available: http://hdl.handle.net/10198/6998..

[45] M. Berón, P. Henriques, M. J. Pereira, and R. Uzal, "Static and dynamic strategies to understand c programs by code annotation," in *1st International Workshop on Fondations and Techniques for Open Source Software Certification*, Braga, 2007.

[46] D. Cruz, M. Berón, P. Henriques, and M. J. Pereira, "Code inspection approaches for program visualization," *Acta Electrotechnica et Informatica*, vol. 9, no. 2, pp. 32–42, 2009, ISSN: 1335-8243.

[47] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, ISSN: 0001-0782. DOI: 10.1145/359545.359563. [Online]. Available: https://doi.org/10.1145/359545.359563.

[48] C. Santilli, *C++ cheat*, https://github.com/cirosantilli/cpp-cheat, Accessed: 2024-05-23, 2024.

[49] D. A. R. Farina, R. Campiolo, J. Rufino, and M. J. V. Pereira, "Automatic and dynamic visualization of process-based concurrent programs," in *Proceedings of the 13th Symposium on Languages, Applications and Technologies (SLATE 2024)*, (Jul. 2024), J. P. Leal, F. Portela, and M. Rodrigues, Eds., ser. OpenAccess Series in Informatics (OASIcs), vol. 120, Águeda, Portugal: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024, pp. 1–12.

Appendix A

Original project proposal

INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

MESTRADO EM INFORMÁTICA

Unidade Curricular de "Dissertação/Projeto/Estágio"

Proposta de tema

☑ Dissertação ☐ Projeto ☐ Estágio

Título

Animação de Programas Paralelos Para Apoio a Aprendizagem

Orientador

Maria João Tinoco Varanda Pereira	mjoao@ipb.pt

Co-orientador IPB

José Carlos Rufino Amaro	rufino@ipb.pt

Co-orientador externo

Rodrigo Campiolo	rcampiolo@utfpr.edu.br

Instituição do Co-orientador externo

Universidade Tecnológica Federal do Paraná	http://www.utfpr.edu.br/

Palavras-chave

Processamento de Linguagens, Compiladores, Animação, Visualização de algoritmos, Programação concorr

Objetivos

O objetivo deste projeto é criar animações para programas escritos em C para auxiliar no ensino de programação concorrente.

Aluno

57060	Daniel Augusto Rodrigues Farina	

Descrição adicional

Atualmente existem algumas ferramentas e plataformas para a visualização da execução, ou o ensino da linguagem de programação C, por exemplo, o Python Tutor e o Log2Base2. Também é possível acompanhar a execução de programas utilizando depuradores de código integrados a uma IDE ou mesmo o depurador gdb no terminal para fazer o acompanhamento da execução. Os primeiros não são capazes de alcançar o nosso objetivo de visualizar a execução de programas utilizando threads. Os depuradores, por outro lado, são capazes de representar a execução de programas concorrentes, mas não são voltados para o ensino.

Neste trabalho pretende-se criar uma ferramenta capaz de auxiliar no aprendizado de programação concorrentes gerando animações da execução de programas escritos em C para uso didático.

Metodologia / Plano de trabalhos

- Estudo do estado da arte (15 dias)
- Desenvolvimento do processamento de programas escritos em C. (90 dias)
- Desenvolvimento da aplicação para gerar as animações. (90 dias)
- Desenvolvimento da integração entre o processamento e as animações. (15 dias)
- Testes da solução (60 dias)
- Validar o aproveitamento do material com alunos da disciplina de programação concorrente. (30 dias)
- Escrita da dissertação (60 dias)

Pré-requisitos

Conhecimento de compiladores;
Conhecimento de linguagem de programação;
Conhecimento de animação;

Recursos necessários

Computador pessoal e acesso à Internet.